Discovering the iOS Instruments Server

Troy Bowman
Hex-Rays

Recon Montreal 2018

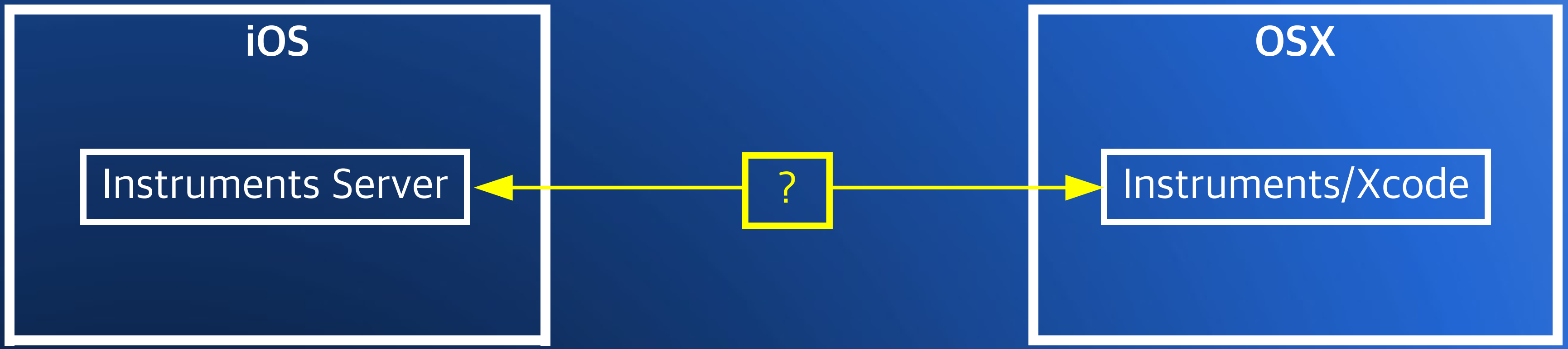
Purpose of This Talk

- Share our discoveries
- Document all of our steps
- Fun!

What is Instruments?

- Instruments is a set of debugging tools developed by Apple:
  - Time Profiling
  - Leak Checking
  - Tracking File I/O
- All of these tasks can be performed on iOS apps as well
- To do this, Apple implements a server that is designed to provide iOS debugging statistics to the Instruments front-end running on OSX
- This server is a goldmine of useful info

What We Want



- How does the server transmit info to OSX? Could IDA interact with it?
- Start with a specific objective: find out how Xcode queries the Instruments server for the process list (i.e. the name of each process running on the target device, along with its PID)

What We Know

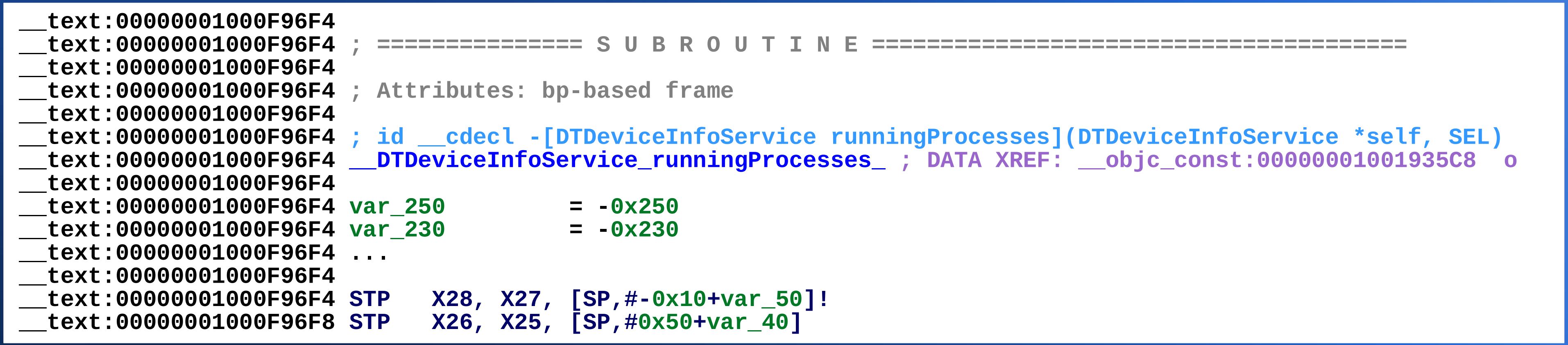
- Somehow, we'll have to communicate with an internal iOS process (normally forbidden on iOS)
- MobileDevice.framework provides ability for OSX apps to communicate with certain iOS processes, a.k.a. “Services”
- This is how IDA communicates with the iOS debugserver
- If we're lucky, there is also a Service for the Instruments server:

  $ hdiutil mount DeveloperDiskImage.dmg

  - com.apple.debugserver.plist
  - com.apple.instruments.remoteserver.plist
  - com.apple.instruments.deviceservice.plist

```
// launch the debugserver service
static bool start_dbgsrv(void *device_handle)
{
  void *srv_handle = NULL;

  mach_error_t err = AMDeviceSecureStartService(
      device_handle,
      CFSTR(“com.apple.debugserver”),
      &srv_handle);

  if ( err != 0 )
    return false;

  char buf[1024];
  size_t nrecv = AMDServiceConnectionReceive(
      srv_handle,
      buf,
      sizeof(buf));

  printf(“received %llx bytes\n”, nrecv);
  return true;
}
```

DVTInstrumentsFoundation

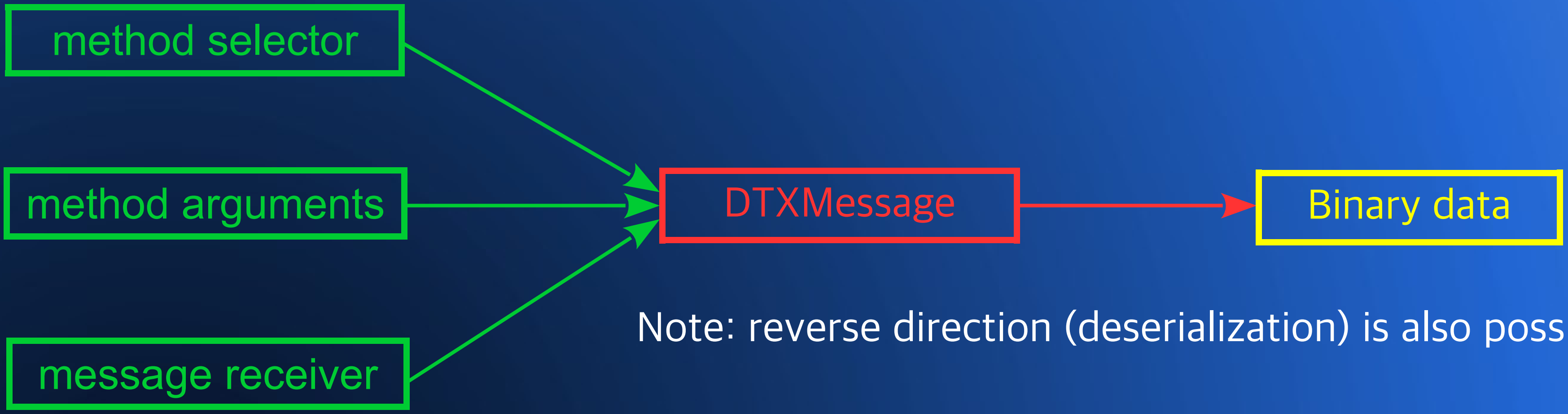


```
__text:00000001000F96F4
__text:00000001000F96F4 ; =============== S U B R O U T I N E =======================================
__text:00000001000F96F4
__text:00000001000F96F4 ; Attributes: bp-based frame
__text:00000001000F96F4
__text:00000001000F96F4 ; id __cdecl -[DTDeviceInfoService runningProcesses](DTDeviceInfoService *self, SEL)
__text:00000001000F96F4 __DTDeviceInfoService_runningProcesses_ ; DATA XREF: __objc_const:00000001001935C8  o
__text:00000001000F96F4
__text:00000001000F96F4 var_250         = -0x250
__text:00000001000F96F4 var_230         = -0x230
__text:00000001000F96F4 ...
__text:00000001000F96F4
__text:00000001000F96F4 STP   X28, X27, [SP,#-0x10+var_50]!
__text:00000001000F96F8 STP   X26, X25, [SP,#0x50+var_40]
```

- Binaries DTServiceHub, DVTInstrumentsFoundation implement core functionality of the Instruments server
- In DVTInstrumentsFoundation we find an interesting method: -[DTDeviceInfoService runningProcesses]
- This method calls +[NSProcessInfo processInfo], then populates an NSMutableArray with a description of each process, and returns it
- Great! It looks like we are on the right track. But…
  - There are no xrefs to this method
  - Selector “runningProcesses” appears nowhere else in the iOS developer tools
  - Who calls it?? A stack trace would help…

DTXMessage

- DTXMessage class is responsible for serializing/deserializing Objective-C messages
- An instance of this class can encode all the information necessary to call a given Objective-C method:



- The Instruments server is invoking critical logic by reading serialized messages from a buffer. What is the source of this serialized data?

DTXMessage

One thread reads serialized messages from a buffer:

```
00000001816A1014    libsystem_kernel.dylib      _semaphore_wait_trap+8
000000018157E3E4    libdispatch.dylib           __dispatch_semaphore_wait_slow+F0
00000001000C8C4C    DTXConnectionServices       -[DTXMessageParser waitForMoreData:incrementalBuffer:]+68
00000001000C87A4    DTXConnectionServices       -[DTXMessageParser parseMessageWithExceptionHandler:]+40
00000001000C8510    DTXConnectionServices       -[DTXMessageParser initWithMessageHandler:andParseExceptionHandler:]_block_invoke+24
000000018156D4B8    libdispatch.dylib           __dispatch_call_block_and_release+14
```

After each message is parsed, another thread is dispatched to perform the invocation:

```
00000001001096F4    DVTInstrumentsFoundation    id __cdecl -[DTDeviceInfoService runningProcesses](DTDeviceInfoService *self, SEL)
0000000181B28ADC    CoreFoundation              ___invoking___+8C
0000000181A20544    CoreFoundation              -[NSInvocation invoke]+118
00000001000CD3D0    DTXConnectionServices       -[DTXMessage invokeWithTarget:replyChannel:validator:]+2C8
00000001000C49980   DTXConnectionServices       -[DTXChannel _scheduleMessage:tracker:withHandler:]_block_invoke697+7C
000000018156D4B8    libdispatch.dylib           __dispatch_call_block_and_release+14
```

DTXMessage

Browsing through the Xcode IDE binaries, we find this in IDEiPhoneSupport:

```
// simplified +[DVTiPhoneProcessInformation requestCurrentProcessInformationsForDevice:usePairedDevice:resultHandler:]
void __cdecl +[DVTiPhoneProcessInformation requestCurrentProcessInformationsForDevice:](id self, SEL a2, id a3)
{
  id v1;
  id v2;
  Block_layout_5803 v3;

  v1 = objc_msgSend(a3, "primaryInstrumentsServer");
  v2 = objc_msgSend(v1, "makeChannelWithIdentifier:", CFSTR("com.apple.instruments.server.services.deviceinfo"));
  if ( v2 )
  {
    v3.isa = _NSConcreteStackBlock;
    v3.flags = 0xC2000000;
    v3.reserved = 0;
    v3.invoke = requestCurrentProcessInformationsForDevice_block_invoke;
    v3.descriptor = &__block_descriptor_tmp_62;
    v3.lvar1 = objc_retain(v2);
    objc_msgSend(&OBJC_CLASS___DVTFuture, "futureWithBlock:", &v3);
  }
}

// call -[DTDeviceInfoService runningProcesses] in the Instruments server process
void __cdecl requestCurrentProcessInformationsForDevice_block_invoke(Block_layout_5803 *a1)
{
  id v1;

  v1 = objc_msgSend(&OBJC_CLASS___DTXMessage, "messageWithSelector:objectArguments:", "runningProcesses", NULL);
  objc_msgSend(a1->lvar1, "sendControlAsync:replyHandler:", v1, NULL);
}
```

found the selector of our favourite method. curious...

DTXMessage

- This critical piece of code is actually a decompiled block function. The original source would probably look something like:

```
@implementation DVTiPhoneProcessInformation
+ (void) requestCurrentProcessInformationForDevice:(id)device
{
  id server  = [device primaryInstrumentsServer];
  id channel = [server makeChannelWithIdentifier:@"com.apple.instruments.server.services.deviceinfo"];
  if ( channel )
  {
    [DVTFuture futureWithBlock:^{
      [channel sendControlAsync:[DTXMessage messageWithSelector:"runningProcesses" objectArguments:Nil]];
    }];
  }
}
@end
```

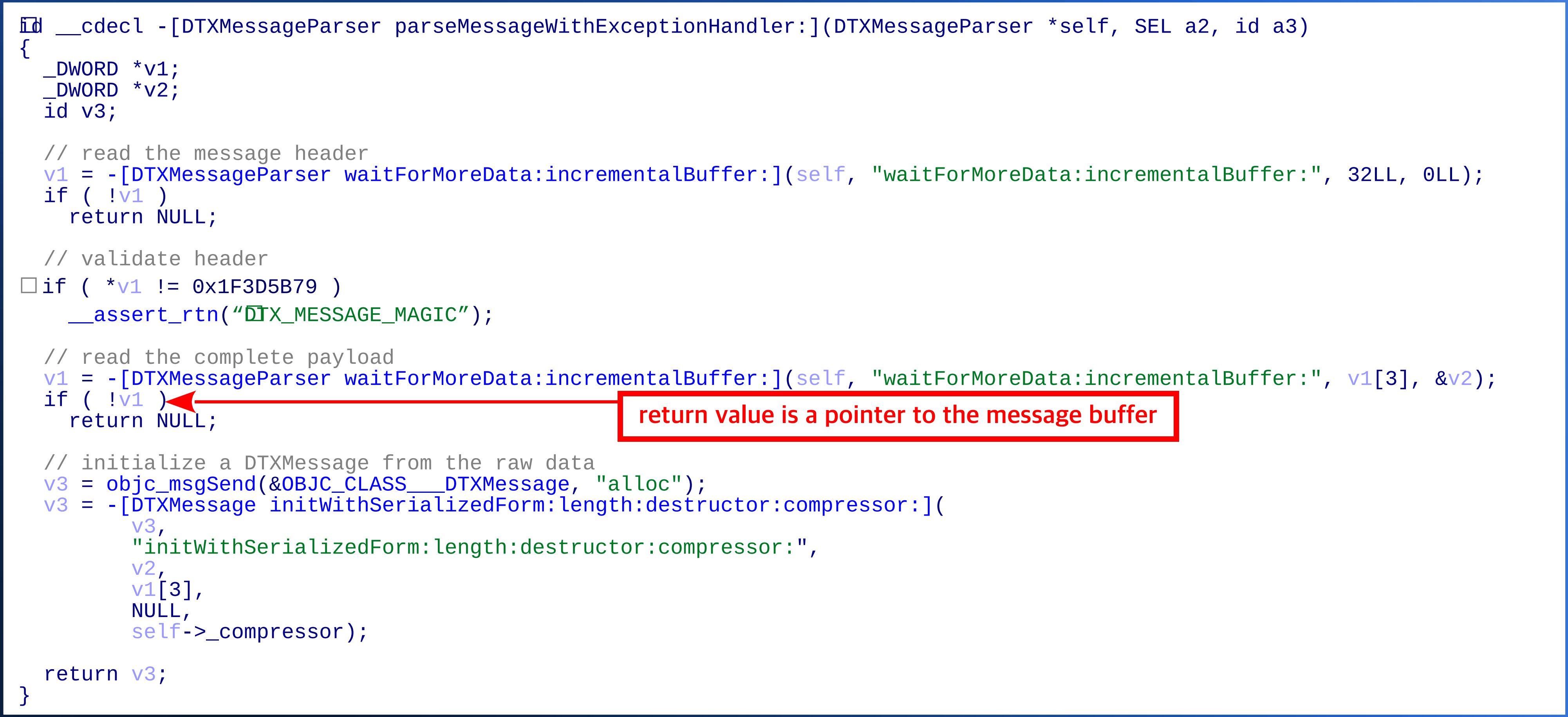
- Conclusion: DTXMessage is a mechanism for transmitting Objective-C messages over the network. This allows a process to effectively “call” a given method in another process (that could be running on a totally separate device)

DTXMessage

- Now what?
- It is possible to reverse-engineer the format of serialized Objective-C messages from the disassembly, but this is an uphill battle
- Even if we understand message serialization perfectly, it still doesn't tell the whole story. What if the server only responds to a specific sequence of messages?
- An alternate approach would be to obtain samples of the raw data transmitted over the wire. A static + dynamic approach has a higher probability of success.
- Again, iOS debugger to the rescue!

DTXMessage

Specially placed breakpoints would allow us to log each serialized message received by the server:



```
id __cdecl -[DTXMessageParser parseMessageWithExceptionHandler:](DTXMessageParser *self, SEL a2, id a3)
{
  _DWORD *v1;
  _DWORD *v2;
  id v3;

  // read the message header
  v1 = -[DTXMessageParser waitForMoreData:incrementalBuffer:](self, "waitForMoreData:incrementalBuffer:", 32LL, 0LL);
  if ( !v1 )
    return NULL;

  // validate header
□if ( *v1 != 0x1F3D5B79 )
    __assert_rtn("DTX_MESSAGE_MAGIC");

  // read the complete payload
  v1 = -[DTXMessageParser waitForMoreData:incrementalBuffer:](self, "waitForMoreData:incrementalBuffer:", v1[3], &v2);
  if ( !v1 )
    return NULL;

  // initialize a DTXMessage from the raw data
  v3 = objc_msgSend(&OBJC_CLASS___DTXMessage, "alloc");
  v3 = -[DTXMessage initWithSerializedForm:length:destructor:compressor:](
         v3,
         "initWithSerializedForm:length:destructor:compressor:",
         v2,
         v1[3],
         NULL,
         self->_compressor);

  return v3;
}
```

return value is a pointer to the message buffer

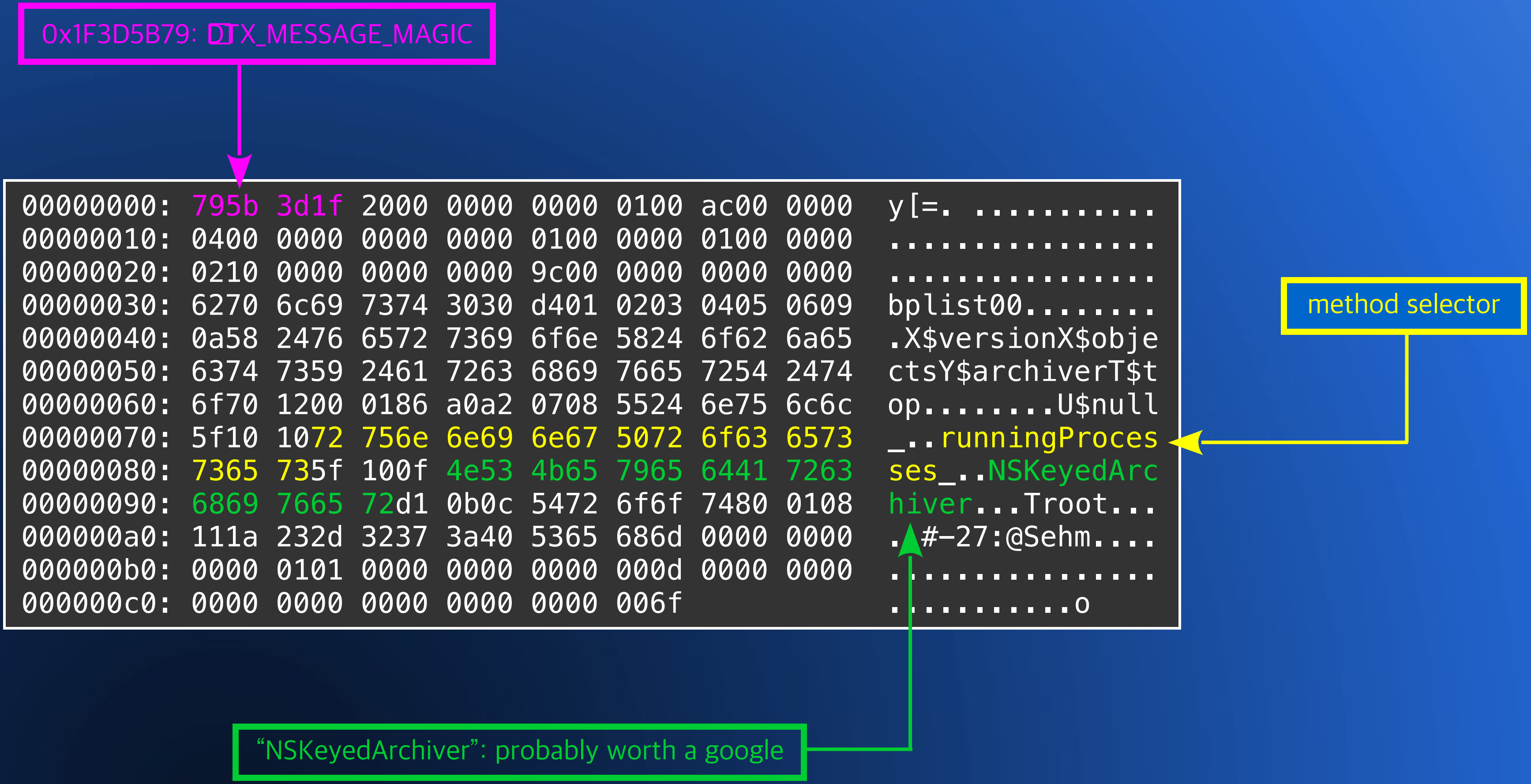
dtxmsg

- I developed an IDA plugin to log the messages automatically:

  https://github.com/troybowman/dtxmsg

- The plugin uses the decompiler's microcode to detect where and when the serialized messages will be available in memory, and dumps the bytes to a file
- For more on the microcode, see Ilfak's talk at Recon Brussels 2018
- The plugin can also deserialize each intercepted message and print the payload to a file in plain text (more on this later)

Anatomy of a DTXMessage


0x1F3D5B79: DTX_MESSAGE_MAGIC
method selector
“NSKeyedArchiver”: probably worth a google

```
00000000: 795b 3d1f 2000 0000 0000 0100 ac00 0000  y[=. ...........
00000010: 0400 0000 0000 0000 0100 0000 0100 0000  ................
00000020: 0210 0000 0000 0000 9c00 0000 0000 0000  ................
00000030: 6270 6c69 7374 3030 d401 0203 0405 0609  bplist00........
00000040: 0a58 2476 6572 7369 6f6e 5824 6f62 6a65  .X$versionX$obje
00000050: 6374 7359 2461 7263 6869 7665 7254 2474  ctsY$archiverT$t
00000060: 6f70 1200 0186 a0a2 0708 5524 6e75 6c6c  op........U$null
00000070: 5f10 1072 756e 6e69 6e67 5072 6f63 6573  _..runningProces
00000080: 7365 735f 100f 4e53 4b65 7965 6441 7263  ses_..NSKeyedArc
00000090: 6869 7665 72d1 0b0c 5472 6f6f 7480 0108  hiver...Troot...
000000a0: 111a 232d 3237 3a40 5365 686d 0000 0000  ..#-27:@Sehm....
000000b0: 0000 0101 0000 0000 0000 000d 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 006f            ...........o
```

Anatomy of a DTXMessage

It looks like the method selector is serialized using an NSKeyedArchiver, but what about the method arguments?

installedApplicationsMatching:registerUpdateToken:
(accepts two arguments)

runningProcesses takes no arguments, so lets look at a different message:

```
00000000: 795b 3d1f 2000 0000 0000 0100 6d02 0000  y[=. .......m...
...
000000c0: 6d65 5824 636c 6173 7365 735c 4e53 4469  meX$classes\NSDi
000000d0: 6374 696f 6e61 7279 a212 1458 4e53 4f62  ctionary...XNSOb
000000e0: 6a65 6374 5f10 0f4e 534b 6579 6564 4172  ject_..NSKeyedAr
000000f0: 6368 6976 6572 d117 1854 726f 6f74 8001  chiver...Troot..
00000100: 0811 1a23 2d32 373b 4148 505b 6263 6466  ...#-27;AHP[bcdf
...
00000160: 246f 626a 6563 7473 5924 6172 6368 6976  $objectsY$archiv
00000170: 6572 5424 746f 7012 0001 86a0 a207 0855  erT$top........U
00000180: 246e 756c 6c50 5f10 0f4e 534b 6579 6564  $nullP_..NSKeyed
00000190: 4172 6368 6976 6572 d10b 0c54 726f 6f74  Archiver...Troot
000001a0: 8001 0811 1a23 2d32 373a 4041 5356 5b00  .....#-27:@ASV[.
...
00000200: 7012 0001 86a0 a207 0855 246e 756c 6c5f  p........U$null_
00000210: 1032 696e 7374 616c 6c65 6441 7070 6c69  .2installedAppli
00000220: 6361 7469 6f6e 734d 6174 6368 696e 673a  cationsMatching:
00000230: 7265 6769 7374 6572 5570 6461 7465 546f  registerUpdateTo
00000240: 6b65 6e3a 5f10 0f4e 534b 6579 6564 4172  ken:_..NSKeyedAr
00000250: 6368 6976 6572 d10b 0c54 726f 6f74 8001  chiver...Troot..
```

archived object (argument 1?)

archived object (argument 2?)

Anatomy of a DTXMessage

The picture is coming into focus, but there's still a missing link: the message receiver.
Who decides which object will receive the message?

Recall that we noticed something like this in Xcode:

```
id channel = [server makeChannelWithIdentifier:@"com.apple.instruments.server.services.deviceinfo"];
[channel sendControlAsync:[DTXMessage messageWithSelector:"runningProcesses" objectArguments:Nil]];
```

-[DTXConnection makeChannelWithIdentifier:] seems to determine the message receiver.
What does this method do? Note that we captured this message:

```
00000000: 795b 3d1f 2000 0000 0000 0100 a301 0000  y[=. ...........
...
00000090: 0708 5524 6e75 6c6c 5f10 3063 6f6d 2e61  ..U$null_.0com.a
000000a0: 7070 6c65 2e69 6e73 7472 756d 656e 7473  pple.instruments
000000b0: 2e73 6572 7665 722e 7365 7276 6963 6573  .server.services
000000c0: 2e64 6576 6963 6569 6e66 6f5f 100f 4e53  .deviceinfo_..NS
000000d0: 4b65 7965 6441 7263 6869 7665 72d1 0b0c  KeyedArchiver...
...
00000150: 6e75 6c6c 5f10 235f 7265 7175 6573 7443  null_.#_requestC
00000160: 6861 6e6e 656c 5769 7468 436f 6465 3a69  hannelWithCode:i
00000170: 6465 6e74 6966 6965 723a 5f10 0f4e 534b  dentifier:_..NSK
00000180: 6579 6564 4172 6368 6976 6572 d10b 0c54  eyedArchiver...T
00000190: 726f 6f74 8001 0811 1a23 2d32 373a 4066  root.....#-27:@f
```

com.apple.instruments.server.services.deviceinfo

requestChannelWithCode:identifier:

dtxmsg: Synopsis

- Let's summarize:

  When querying the process list, Xcode sent 5 messages to the Instruments server:

  - _notifyOfPublishedCapabilities:
  - _requestChannelWithCode:identifier:
    - identifier = "com.apple.instruments.server.services.deviceinfo"
  - _requestChannelWithCode:identifier
    - identifier = "com.apple.instruments.server.services.device.applictionListing”
  - runningProcesses
  - installedApplicationsMatching:registerUpdateToken:

dtxmsg: Synopsis

- Let's summarize:

  When querying the process list, Xcode sent 5 messages to the Instruments server:

  - _notifyOfPublishedCapabilities:
  - _requestChannelWithCode:identifier:
    - identifier = "com.apple.instruments.server.services.deviceinfo"
  - _requestChannelWithCode:identifier
    - identifier = "com.apple.instruments.server.services.device.applictionListing”
  - runningProcesses
  - installedApplicationsMatching:registerUpdateToken:

* These messages request a list of all installed apps. Interesting, but not absolutely necessary.

dtxmsg: Synopsis

- Let's summarize:

  When querying the process list, Xcode sent 5 messages to the Instruments server:

  - _notifyOfPublishedCapabilities:
  - _requestChannelWithCode:identifier:
    - identifier = "com.apple.instruments.server.services.deviceinfo"
  - _requestChannelWithCode:identifier
    - identifier = "com.apple.instruments.server.services.device.applictionListing”
  - runningProcesses
  - installedApplicationsMatching:registerUpdateToken:

* Likely the minimal required behaviour for querying the proclist

Final Steps: Decompilation

- Remember that our goal is to communicate with the server independently, without the assistance of Apple's code
- Our understanding of serialized messages must be perfect
- Fortunately we already have a lot of clues
- IDA is invaluable here. We can aggressively refine the decompilation for all of the critical methods we've found so far.

Final Steps: Decompilation

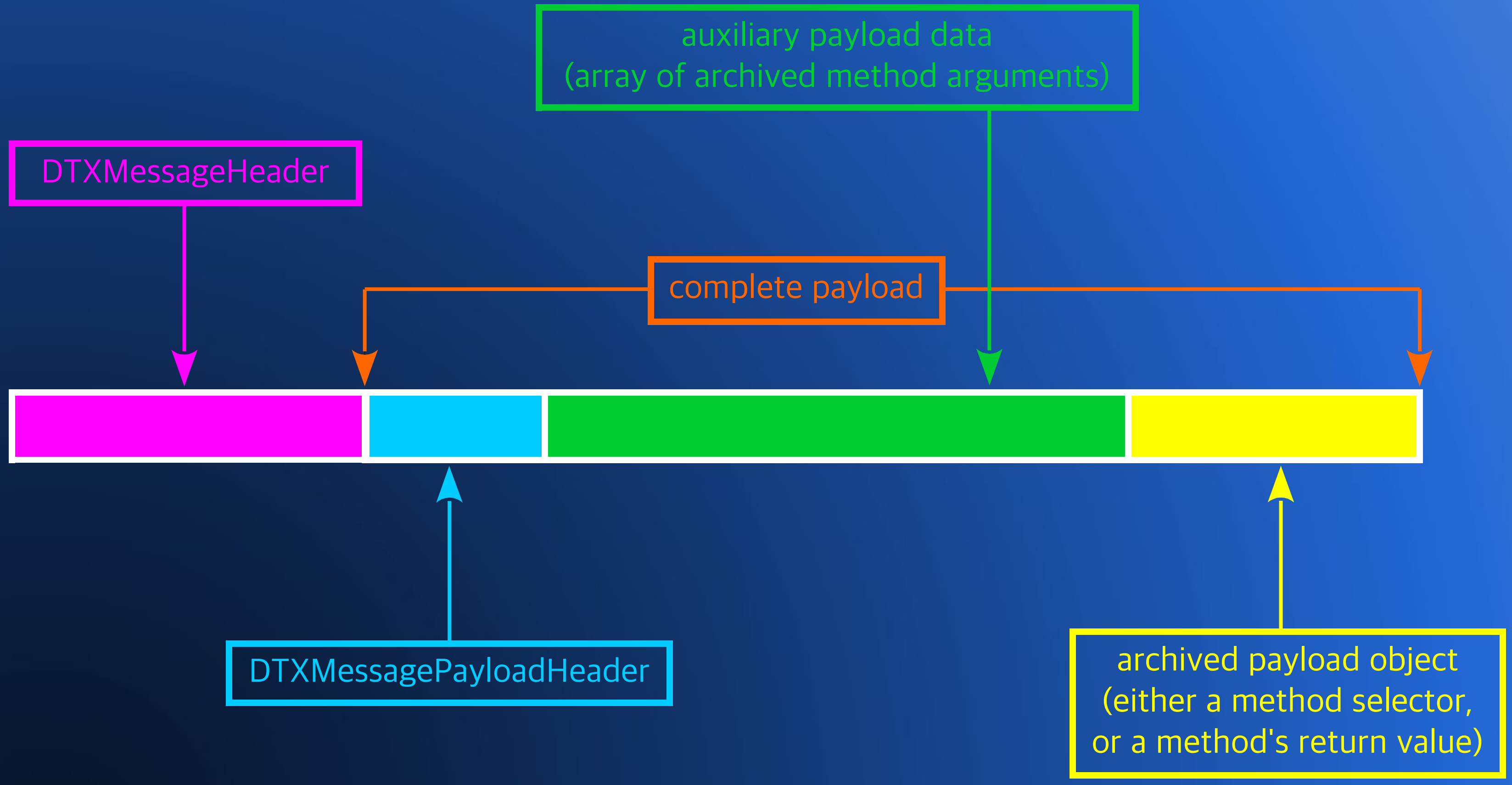
Decompilation yields some important structures:

```
// a DTXMessage object in memory
struct DTXMessage
{
  NSObject super;
  int32 _messageType;
  int32 _compressionType;
  uint32 _status;
  id _destructor;
  const char *_internalBuffer;
  uint64 _internalBufferLength;
  uint64 _cost;
  id _payloadObject;
  void *_auxiliary;
  bool _deserialized;
  bool _immutable;
  bool _expectsReply;
  uint32 _identifier;
  uint32 _channelCode;
  uint32 _conversationIndex;
  NSDictionary *_auxiliaryPromoted;
};
```

```
// header for serialized message data
struct DTXMessageHeader
{
  uint32 magic;
  uint32 cb;
  uint16 fragmentId;
  uint16 fragmentCount;
  uint32 length;
  uint32 identifier;
  uint32 conversationIndex;
  uint32 channelCode;
  uint32 expectsReply;
};
```

```
// layout of serialized payload
struct DTXMessagePayloadHeader
{
  uint32 flags;
  uint32 auxiliaryLength;
  uint64 totalLength;
};
```

Anatomy of a DTXMessage: Finalized



dtxmsg_client

- We're finally ready to start sending messages ourselves
- A standalone application (dtxmsg_client) is also included with the dtxmsg plugin source
- This app is able to invoke the runningProcesses method, retrieve its return value, and print it in plain text
- Objective complete!
- Source code for this app is available for reference

dtxmsg_client

- Extra credit:

  There are some other methods that might be of interest to us:

  - -[DTApplicationListingService installedApplicationsMatching:registerUpdateToken:]
  - -[DTProcessControlService launchSuspendedProcessWithDevicePath:bundleIdentifier:environment:arguments:]
  - -[DTProcessControlService killPid:]
- dtxmsg_client can invoke all of these methods as well
- They provide a little more insight into how complex method arguments and return values are handled

Future Work

- The Instruments server is responsible for much more than just simple process control
- DTXMessage is also used by other iOS developer tools
- Hopefully this is just the beginning!
- Thanks for your time