iOS Automation Primitives

OWASP

The Open Web Application Security Project

April 13, 2016

iOS Automation Primitives

(Hacking in context)

Mikhail Sosonkin

mikhail@synack.com http://debugtrap.com

Security Researcher at SYNACK

Working on low level emulation with QEMU and iPhone automation.

Graduate of Polytechnic University/ISIS Lab 2005

a.k.a New York University Tandon School of Engineering

Masters in Software Engineering from Oxford University 2014

Exeter College

СССР 1986

Intel 8080 Clone

1.78MHz CPU

32KB RAM

2KB ROM

450 Rubles

Wikipedia-RU

Why automation?

Time saving

More thorough

Repeatable

API Discovery

Code Coverage

Discover Preinstalled Malware

Cameras arrived with malware from Amazon

"When you automate tests of UI interactions, you free critical staff and resources for other work." - Apple

Getting started with iOS

- Get iPhone 5s
  - Swappa
- Apply Jailbreak
  - Install OpenSSH via Cydia
  - Use tcprelay to SSH over USB
- Start exploring
  - Debugserver
- Objective-c: Phrack 0x42
  - http://phrack.org/issues/66/4.html
- iOS App Reverse Engineering

  The world's 1st book of very detailed iOS App reverse engineering skills :)
- TCP Relay

Pangu

TaiG

The goal

“We want to dissect and study an application that we have no developer control over”

Static Analysis

- Use dumpdecrypted by Stefan Esser to acquire the binary
- IDAPro for reverse engineering
- Class-dump to get the Objective-C meta data.
    - Objective-C is automation's best friend

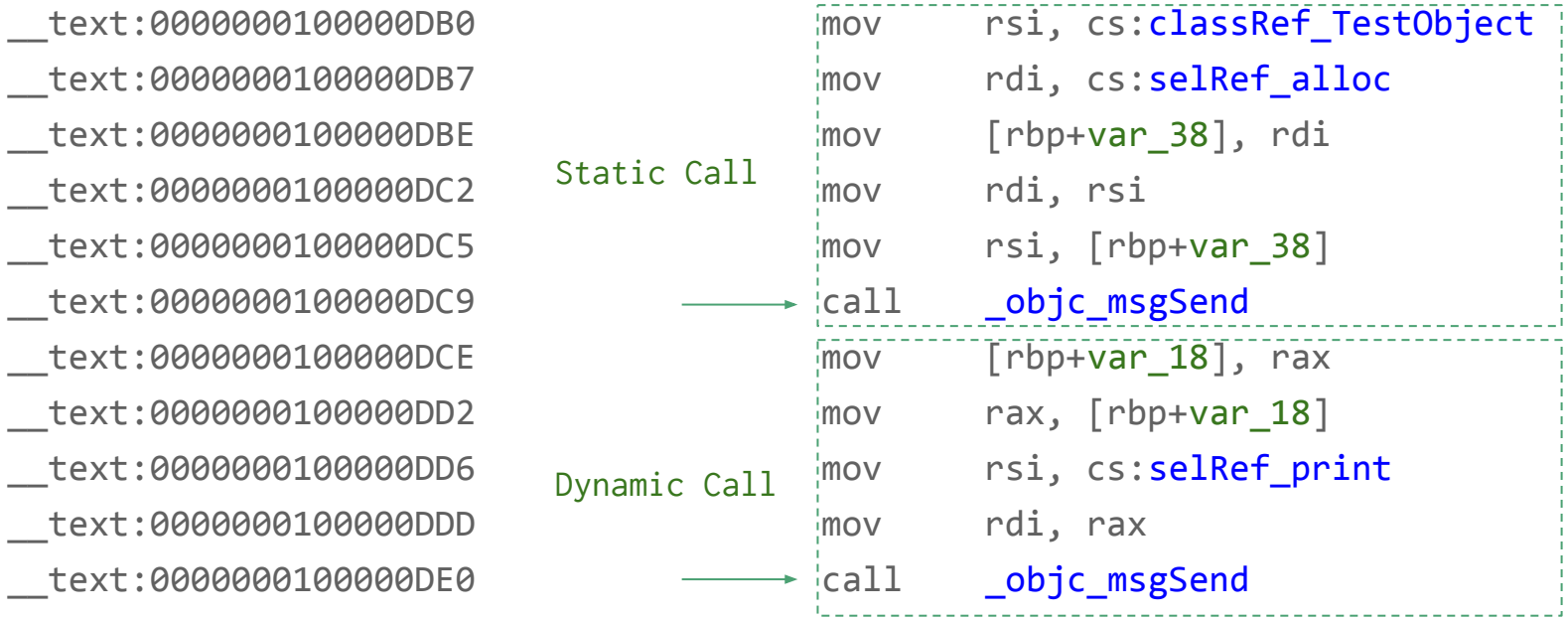
Let’s explorer how Objective-C

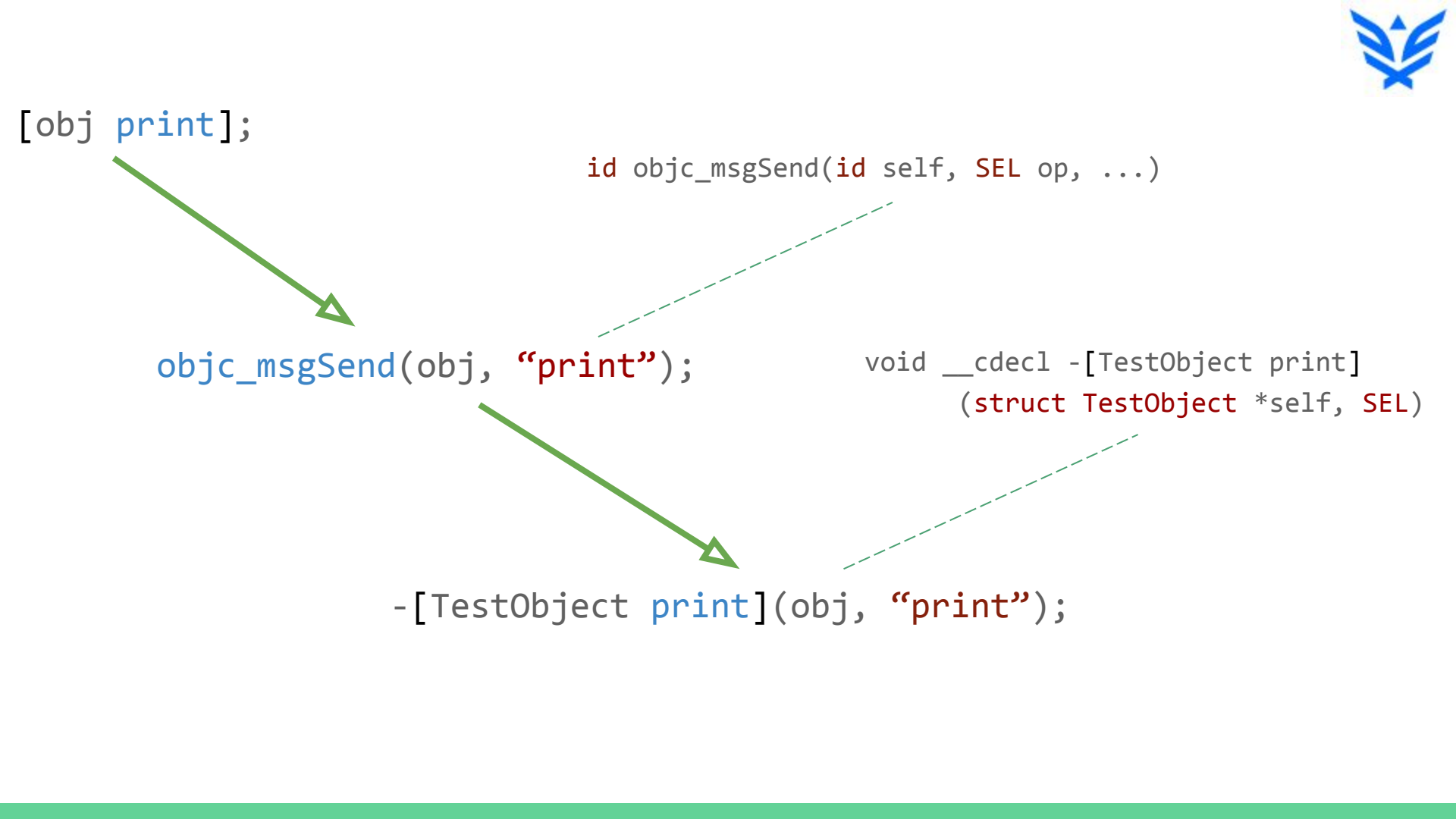
calls methods

```
@interface TestObject : NSObject {    }

  -(void)print;

@end

@implementation TestObject

  -(void)print {     NSLog(@"Test Object");     }

@end
```

...

```
TestObject* obj = [TestObject alloc];
[obj print];
```


__text:0000000100000DB0 mov rsi, cs:classRef_TestObject
__text:0000000100000DB7 mov rdi, cs:selRef_alloc
__text:0000000100000DBE mov [rbp+var_38], rdi
Static Call
__text:0000000100000DC2 mov rdi, rsi
__text:0000000100000DC5 mov rsi, [rbp+var_38]
__text:0000000100000DC9 call _objc_msgSend
__text:0000000100000DCE mov [rbp+var_18], rax
__text:0000000100000DD2 mov rax, [rbp+var_18]
__text:0000000100000DD6 mov rsi, cs:selRef_print
Dynamic Call
__text:0000000100000DDD mov rdi, rax
__text:0000000100000DE0 call _objc_msgSend


[obj print];
id objc_msgSend(id self, SEL op, ...)
objc_msgSend(obj, “print”);
void __cdecl -[TestObject print]
(struct TestObject *self, SEL)
-[TestObject print](obj, “print”);

Dynamic Analysis

- Verbose nature of Objective-C
  - Query Objects
  - Trigger method calls
- Debugging
  - Cycript
  - Frida
  - Custom DYLIB
- Injecting into the App
  - MobileSubstrate
  - DYLD_INSERT_LIBRARIES

Dynamic tools

- Frida
  - Binary Instrumentation using JavaScript
  - Mostly for debugging and tracing

- Cycript
  - Injectable debugger
  - Manipulate and examine objects
  - iOS Spelunking (Talk and OWASP NYC)
    - Showing how to rewire an application to discover more.

Network tools

- MITMProxy
  - Intercept network data
  - Write custom scripts for transformations

- iOS Disable Certificate pinning
  - https://github.com/iSECPartners/ios-ssl-kill-switch
  - **WARNING: THIS TWEAK WILL MAKE YOUR DEVICE INSECURE** 

Available Frameworks

"Appium is an open source test automation framework for use with native, hybrid and mobile web apps. It drives iOS and Android apps using the WebDriver protocol." - Appium

"You can use the Automation instrument to automate user interface tests in your iOS app through test scripts that you write." - Apple UI Instruments

All frameworks require you to be the app developer!

Not nice for blackbox testing.

Jailbreakers to the rescue!

So, you want to roll your own?

- Simulate the user

- Read and understand the UI

Generating Events

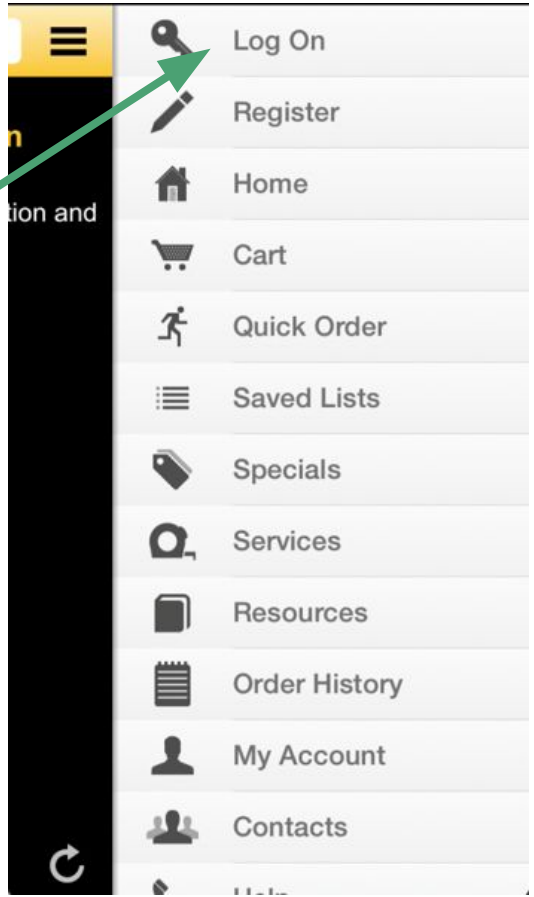
- SimulateTouch
  - http://api.iolate.kr/simulatetouch/
  - Generate TouchUp/TouchDown
  - Generate Swipes
- SimulateKeyboard
  - https://github.com/iolate/SimulateKeyboard
  - Generate Key presses
  - Mechanical and Virtual

Reading the UI

- UIView
  - The source of everything
    - Stems from **UIApp.keyWindow**
  - Constructs a tree structure
  - UILabel
  - UIButton
  - UITextField
  - etc.

Let’s peek in

UILabel and UIButton in a UIScrollView



Sneaking a peek

```
cy# UIApp.keyWindow

<UIWindow; frame = (0 0; 320 568); gestureRecognizers = <NSArray>;>
   | <TiRootViewNeue; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer>>
...
<TiUITableViewCell; baseClass = UITableViewCell; text = 'Log On';
| <TiGradientLayer;> (layer)
| <UITableViewCellContentView; frame = (0 0; 256 43.5); layer = <CALayer>>
|    | <UITableViewLabel; frame = (74 0; 167 43.5); text = 'Log On'>
|    | <UIImageView; frame = (15 0; 44 43.5); layer = <CALayer>>
| <_UITableViewCellSeparatorView; frame = (74 43.5; 182 0.5); layer = <CALayer>>
```

Putting it all together

"An engine for driving the UI while doing blackbox testing of an iOS App"

– **CHAOTICMARCH (On github)**

CHAOTICMARCH

- **Lua Scriptable Logic**
- Standard functions for touching the device
- Options for record/replay
- **Finding UI Components**
- Regulating speed of execution
- Support for multiple targets
- **Mechanisms for generic logic**
- Lightweight injected module

“Lua is a powerful, fast, lightweight, embeddable scripting language … means "Moon" in Portuguese … Please do not write it as "LUA", which is both ugly and confusing”

lua.org

Lua Layout

```
lua
├── chaotic_march.lua
├── com.gs.pwm.external-1-login.lua
├── com.hdsupply.hdsupplyfm-1-search.lua
├── post_all-click_around.lua
├── pre_all-common.lua
└── pre_all-wait_to_start.lua
```

Initialization

1. Dylib reads and executes `chaotic_march.lua`
2. Execute all `pre_all*.lua` scripts
    a. Library functions
    b. Generic logic
3. Execute all `[bundle_id]*.lua`
    a. Target specific logic
4. Execute all `post_all*.lua`
    a. Any sort of common clean up
    b. Close out the execution

CHAOTICMARCH - Target

*“Engine is injected into all apps and so it has to situate **itself**”*

```
getBundleID() ->
    “com.hdsupply.hdsupplyfm”
```

Basic Logic

```
while true do
    local button = getButton(clickedButtons)

    -- put some info in.
    fill_all_fields()
    click_button(button)

    if(button["text"] ~= nil) then
        clickedButtons[button["text"]] = 1
    end
    usleep(2 * 1000000)
end
```

Finding elements

```
local buttons = findOfTypes(
    "UIButton", "UINavigationItemButtonView",
    "UINavigationItemView", "_UIAlertControllerActionView",
    "UISegmentLabel", "UILabel", "")
```

Basically anything we might consider clickable.

Other interesting functions

```
inputText(String text) ->
```

Enter the text into whatever component is holding the focus.

```
hasTextAt(String txt, boxes_x, boxes_y, box_x, box_y) ->
```

Same as component but the engine will look for text at a specified box.

```
findOfTypes(String type1, ..., String "") ->
```

Returns a dictionary of the locations of particular types of components.

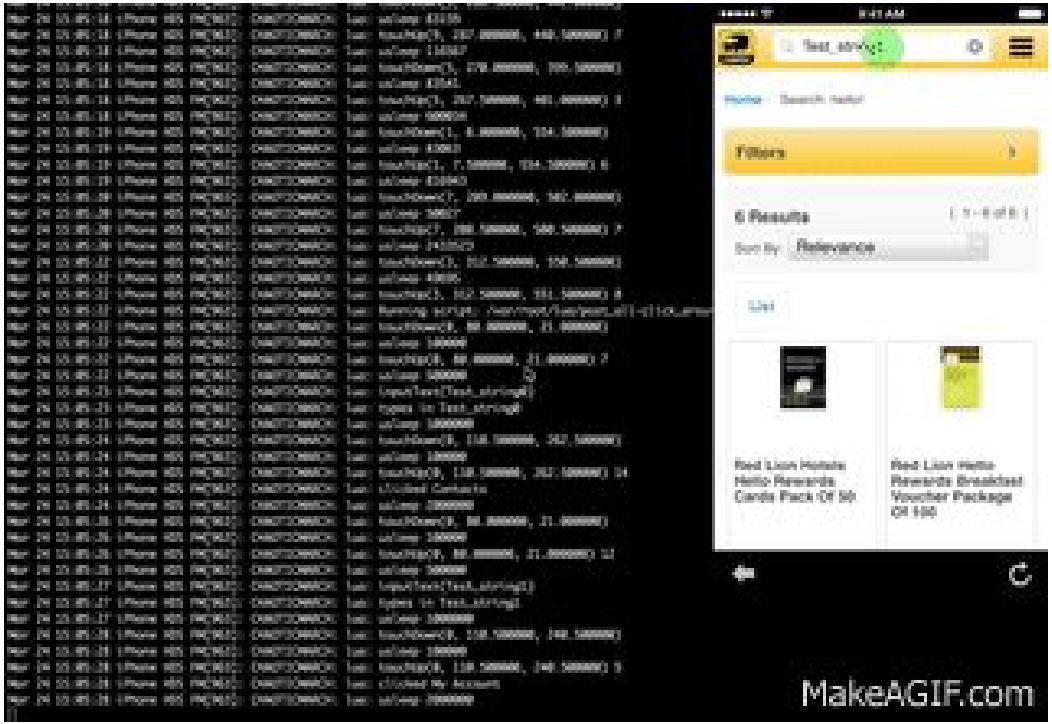
Element representation

```
{
    "x": [x - coordinate, top-left corner],
    "y": [y - coordinate],
    "width": [number],
    "height": [number],
    "text": [best guess at text of the button],
    "title": [Closest title to the element]
}
```

Challenges/Research areas

- Identifying an interesting event
- Recording path to event
- Accurately identifying what the user sees
  - Clickables: Not all are buttons
- Instrumentation
- Repeated triggering
- Handling games and custom UI’s

Demo!



Youtube link

- HD Supply test case
- Replay raw touch
- Fill in forms
- Click buttons

Why?

Together we can build a great library

of testing logic for all kinds of apps!

Thank you!

MIKHAIL SOSONKIN

mikhail@synack.com

https://github.com/synack/chaoticmarch

http://debugtrap.com/