Part 1: Introduction to reverse engineering of iOS applications

Training Title: Mobile Device Security

Twitter: @m_burdach

Table of content

Introduction

This tutorial describes basic methods of reverse engineering of iOS applications. OWASP iGoat mobile application is used in described examples. After several years of working on mobile security projects we can confirm that many errors overlap with errors implemented in iGoat application.

Here is a brief listing of topics discussed in this material:

- Assembly language for ARM64
- Static analysis methods with Hopper
- Dynamic analysis methods with Frida

As mentioned above, the OWASP iGoat mobile application is used in this tutorial. We did small changes in Objective-C version of this application. It is worth to mention that SWIFT version of iGoat application was released a few weeks ago.

ARM64

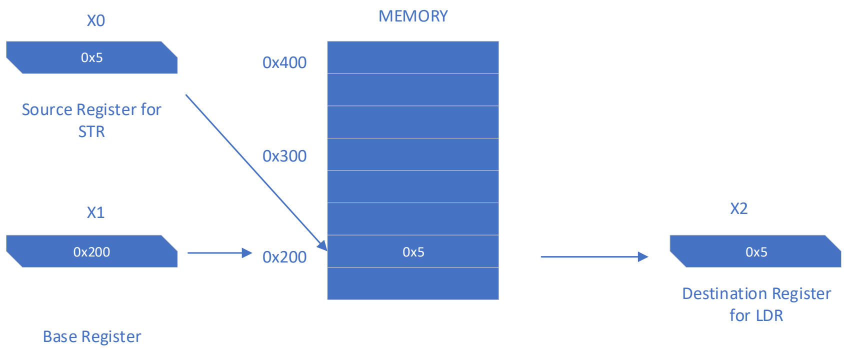
64-bit ARM CPU architecture (AArch64) is used since iPhone 5S. iOS version must be 7.0 or later. In this tutorial we are using only application dedicated for ARM64 CPU.

There are 31 general-propose registers. These registers are labelled x0-x30 in 64-bit mode (in 32-bit mode are named w0-w30). The first 8 registers are used to pass argument values into a subfunction and to return result values from a function. X8-X17 are scratch registers. A subfunction invocation must preserve the contents of the registers R19-R29. Some registers have special roles. For example X29 is the frame pointer register (FR) and X30 is the link pointer register (LR). By default RET instruction is return to address in register X30 (LR). Additionally, a stack pointer register SP exists. Stack pointer is never modified implicitly (there is no pop/push instructions).

Operations on memory are usually performed by using LDR/STR registers. But the other instructions exist – for example LDP/STP to load and store two registers at once.

Below is the simplest example of moving data:

STR X0, [X1] – value at register X0 is stored to address pointed by [X1].
LDR X2, [X1] – value at address pointed by [X1] is loaded into register X2.



Below example contains an additional parameter - the offset.

STR X0, [X1, #12] – value at register X0 is stored to address pointed by [X1 + 0xc]

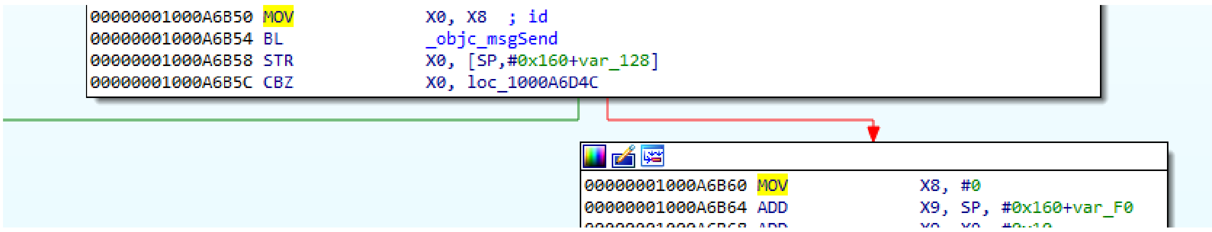
Branch instructions

Branches are PC-relative. It means that the target address is calculated based on the value of the current PC (program counter). The example is B instruction – it is a simple branch – it jumps to PC-relative address. Branch instructions with a l suffix (bl or blx) work like a standard B instruction but also store a return address in the link register (LR). There are several other instruction which control the program flow. For example CBNZ (compare a register and branch on nonzero) or TBNZ (test single bit [defined by immediate byte offset – in range 0 to 63] and branch if nonzero).

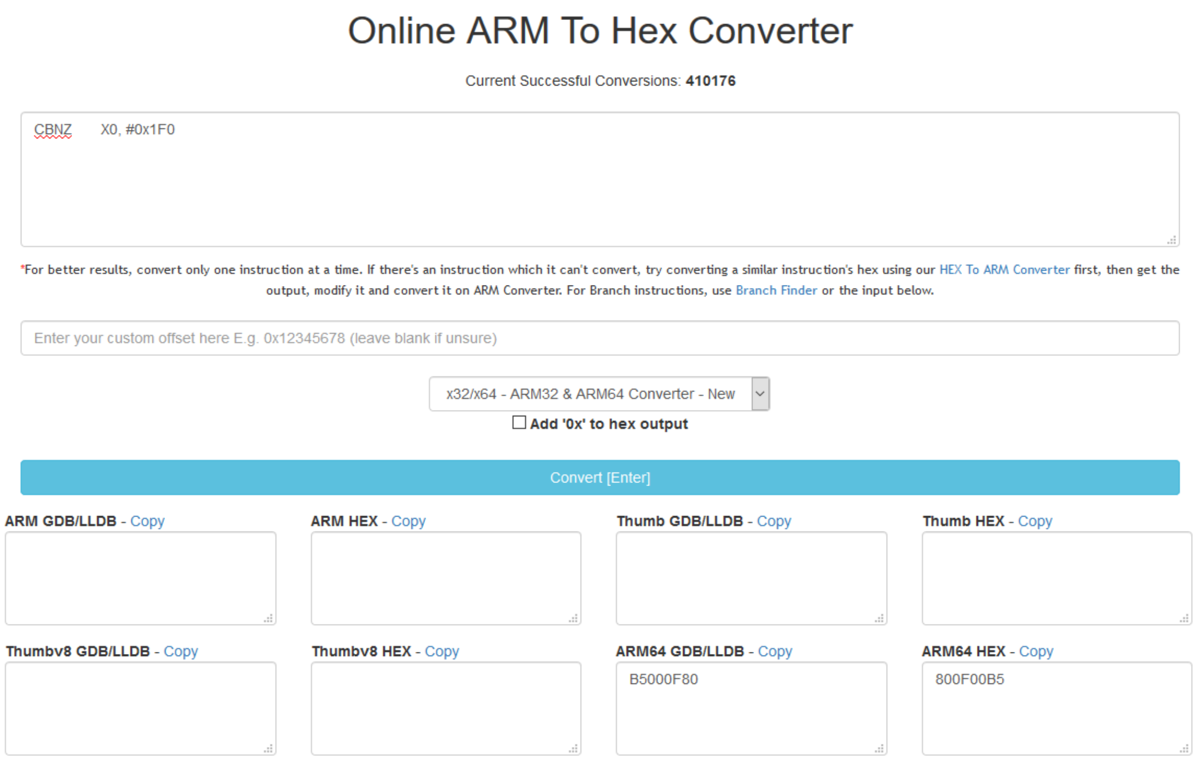
Binary patching

We highly recommend armconverter.com to convert ARM instructions into hex values. Especially when you need to interfere in the program flow.

To change the flow of below code you can modify the condition instruction from CBZ into CBNZ



Hex value for CBZ X0, offset is 80 0F 00 B4. After modification into CBNZ the HEXARM is 80 0F 00 B5. You can patch binary file by using Hopper, IDA Pro or any hex editor.



Frida

Frida is a multiplatform instrumentation / reverse engineering toolkit. Functions delivered by the gum engine are not the same on all platforms – especially for ARM64 but Frida functionality is very helpful during initial analysis of iOS applications.

The goal of this part is to show some of Frida features which are useful during security assessment of iOS applications.

Example #1

Understanding of a program logic is the most important part of every security assessment. Cryptography functions are usually used to hash or encrypt/decrypt sensitive data. In iOS cryptography functions are implemented by libcommonCrypto.dylib library. We can use the Frida-trace tool to check which functions are used by the iGoat app. After creating templates for each exported function frida-trace displays information about called functions (in example below SHA256 and CCDigest).

```
$frida-trace -U iGoat -I libcommonCrypto.dylib
Instrumenting functions…
CCECGetKeyType: Auto-generated handler at "…/__handlers__/libcommonCrypto.dylib/CCECGetKeyType.js"
CC_SHA224_Init: Auto-generated handler at "…/__handlers__/libcommonCrypto.dylib/CC_SHA224_Init.js"
CCCryptorGCMDecrypt: Auto-generated handler at "…/__handlers__/libcommonCrypto.dylib/CCCryptorGCMDecrypt.js"
```

```
...
Started tracing 208 functions. Press Ctrl+C to stop.
        /* TID 0x403 */
 12925 ms  CC_SHA256()
 12925 ms     | CCDigest()
 13875 ms  CC_SHA256()
 13875 ms     | CCDigest()
...
```

Example #2

In next example we would like to display arguments of an internal function. We have to keep in mind that functions implemented in C/C++ will be stripped – it means that function names are not available in disassembled output. The same result is with applications written in Swift. Disassembler (Hopper or IDA Pro) shows us only sub_ name like in example below.

```
================ B E G I N N I N G   O F   P R O C E D U R E ================


                     sub_1000d33d4:
00000001000d33d4         stp        x24, x23, [sp, #-0x40]!                  ; XF
00000001000d33d8         stp        x22, x21, [sp, #0x10]
00000001000d33dc         stp        x20, x19, [sp, #0x20]
00000001000d33e0         stp        x29, x30, [sp, #0x30]
00000001000d33e4         add        x29, sp, #0x30
00000001000d33e8         sub        sp, sp, #0x70
00000001000d33ec         mov        x21, x0
00000001000d33f0         orr        w20, wzr, #0x40
```

Let's create simple script (disassemble.js) to check if this function is available at offset displayed in above example. Frida is using Capstone engine to disassemble code. The findBaseAddress function is critical because the ASLR is used in iOS operating system.

```
var baseAddr = Module.findBaseAddress('ExampleApp');
var orgFuncOffset = 0x000d33d4;
var orgFuncRVA = baseAddr.add(orgFuncOffset);

var a = Instruction.parse(ptr(orgFuncRVA));
var xxy = "ret";
var uua = 1;
var counter = 0; //limit counter of 10 instructions
while(uua != 0 && counter <= 10)
        {
        var bb = Instruction.parse(a.address);
        var uuu = a.mnemonic.toString();
        console.log(a);
        a = Instruction.parse(bb.next);
        uua = xxy.localeCompare(uuu);
        counter++;
}
```

To load our script we have to use –l switch.

```
$ frida -U "ExampleApp" -l disassemble.js

…
Attaching...
stp x24, x23, [sp, #-0x40]!
stp x22, x21, [sp, #0x10]
stp x20, x19, [sp, #0x20]
stp x29, x30, [sp, #0x30]
add x29, sp, #0x30
sub sp, sp, #0x70
mov x21, x0
orr w20, wzr, #0x40
orr w0, wzr, #0x40
bl #0x1000879c8
mov x19, x0
[USB::iPhone::ExampleApp]-> quit
```

I mentioned that function names are not available. But it is true only for C/C++ and Swift. Applications written in Objective-C are using class and method names during execution. So it is very easy to trace such functions in Frida.

Example #3

There are at least 2 ways to display arguments. We can use an array of NativePointer objects or this.context.reg_number (for example this.context.x0). In below example we intercept system API function CCHmac to display arguments like key, input data and output data.

If we would like to display particular registry keys we have to remember that first 8 registers are used to pass argument values to function.

```
var hmacresult
Interceptor.attach(Module.findExportByName('libcommonCrypto.dylib', 'CCHmac'),{
 onEnter: function(args) {
     hmacresult = args[5];
     var keyLength = args[2].toInt32();
     var raw_key = hexdump(args[1], {
        length: keyLength
        });
     var dataLength = args[4].toInt32();
     var raw_data = hexdump(args[3], {
        length: dataLength
        }
        );
     console.log("\nHMAC raw key: " + hexdump(this.context.x1));
     console.log("\nHMAC raw key: " + raw_key + "\n \n");
     console.log("\nHMAC data (SALT): " + raw_data + "\n \n");
     }
        ,onLeave: function (retval) {
        var b2 = Memory.readByteArray(hmacresult, 20);
```

```
        console.log("hmac result: ");
        console.log(hexdump(b2));
        }
});
```

Example #4

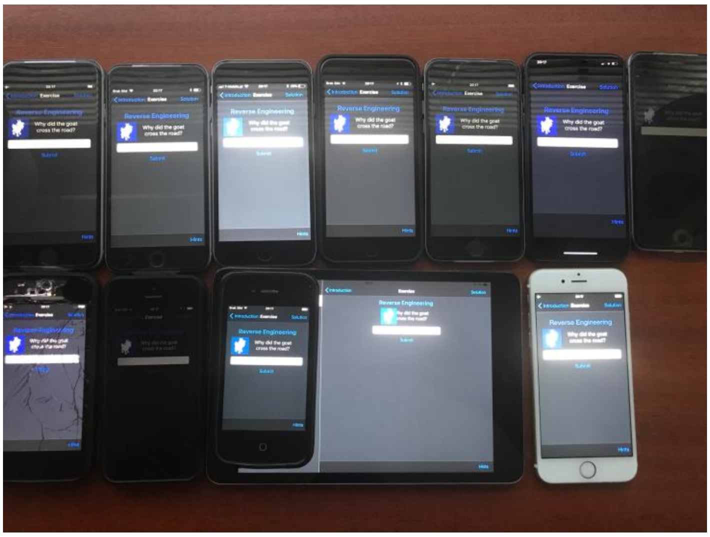
Finally, Frida can be used to perform dynamic binary patching. We can overwrite the argument value directly – for example args[5] = ptr("12345"); It is also possible to replace the return value by using replace function – for example retval.replace(ptr("0x0");
If we do not know where data to replace are stored in memory, we can use the scanSync function. For example to change the userid value we can use the following code:

```
var pattern = " 75 73 65 72 49 64";
var newvalue = "{\"userId\":6666}";
Memory.scanSync(args[6], 15, pattern);
…
Memory.writeUtf8String(args[6], newvalue);
…
```

Practical examples (OWASP iGOAT – Obj-C version)

After reading above information we are ready to start practical exercises. We need a physical device iPhone/iPad with the OWASP iGoat application. Mac OS X with Frida is recommended because we can even run (by using ios-deploy) the OWASP iGoat on non-jailbroken device. Hopper disassembler is only available on Mac OS X and Linux based machines.

Our iGoat "farm" is presented at below picture.

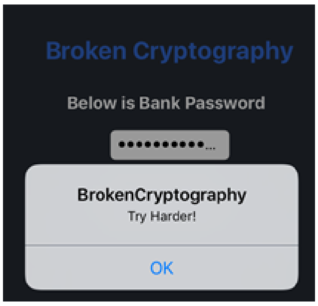


Picture 0x1: OWASP iGoat farm @ prevenity

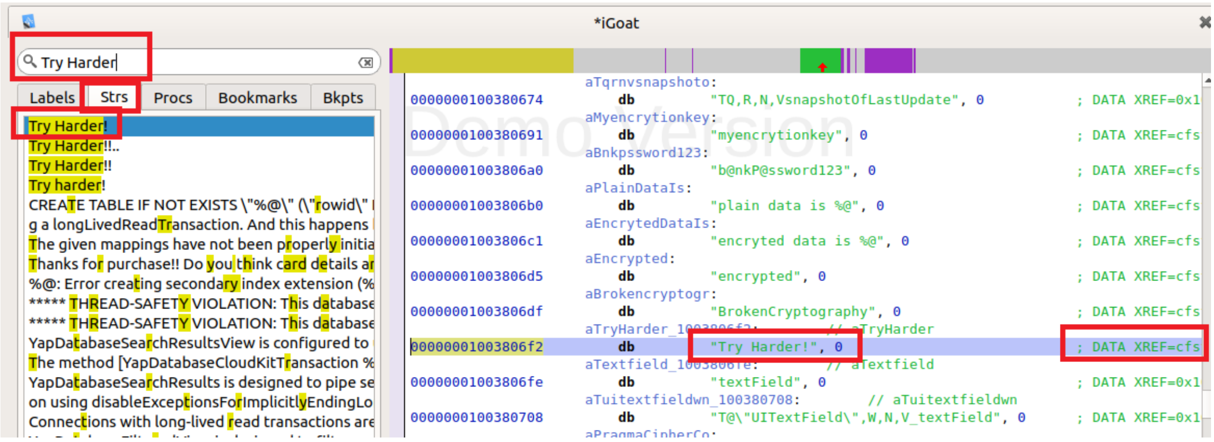
Exercise #1

Exercise Name: **Key Management -> Hardcoded Keys**
Question: How to identify hardcoded encryption key in the OWASP iGoat application?

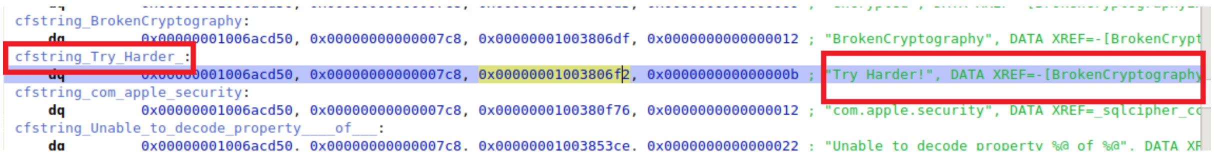
Step 1: After pressing the text „Show Above Text” we can notice the message „Try Harder!” on an iPhone screen.



Step 2: Open the OWASP iGoat binary application (AArch64) in the Hopper disassembler. Then change tab in left panel to Strs (Strings). Retype the message “Try Harder!”.



Step 3: Now, you have to jump to cross reference XREF=cfstring_Try_Harder_ and then jump to DATA XREF. – [BrokenCryptographyExerciseViewController showData:]+48



Step 4: We identified main class for this exercise. Now change the tab in left panel to Procs (Procedures) and type [BrokenCrypto*. Then we have to check every method in class BrokenCryptographyExerciseViewController. Method vidwDidLoad has interesting content. We can identify:

a. Two strings (encryptionKey and password)

```
ldr     x8, [x8, #0xd78] ; 0x100327d78@PAGEOFF, 0x100327d78,__objc_class_BrokenCryptographyExerciseViewController_class
str     x8, [sp, #0x50 + var_38]
adrp    x8, #0x10031f000
ldr     x1, [x8, #0xb70] ; "viewDidLoad",@selector(viewDidLoad)
add     x0, sp, #0x10 ; argument "super" for method imp___stubs__objc_msgSendSuper2
bl      imp___stubs__objc_msgSendSuper2 ; objc_msgSendSuper2
adrp    x8, #0x100328000
ldrsw   x23, [x8, #0xe98] ; objc_ivar_offset_BrokenCryptographyExerciseViewController_encryptionKey
adrp    x20, #0x1002cc000 ; 0x1002ccdf8@PAGE
add     x20, x20, #0xdf8 ; 0x1002ccdf8@PAGEOFF, @"myencrytionkey"
mov     x0, x20
bl      imp___stubs__objc_retain ; objc_retain
ldr     x0, [x19, x23]
str     x20, [x19, x23]
bl      imp___stubs__objc_release ; objc_release
adrp    x8, #0x100323000 ; &@selector(pageTable_insertForPageKeyStatement)
ldr     x20, [x8, #0x468] ; "textField",@selector(textField)
mov     x0, x19
mov     x1, x20
bl      imp___stubs__objc_msgSend ; objc_msgSend
mov     x29, x29
bl      imp___stubs__objc_retainAutoreleasedReturnValue ; objc_retainAutoreleasedReturnValue
mov     x21, x0
adrp    x8, #0x10031f000
ldr     x1, [x8, #0xbb0] ; "setText:",@selector(setText:)
adrp    x2, #0x1002cc000 ; 0x1002cce18@PAGE
add     x2, x2, #0xe18 ; 0x1002cce18@PAGEOFF, @"b@nkP@ssword123"
bl      imp___stubs__objc_msgSend ; objc_msgSend
mov     x0, x21
bl      imp___stubs__objc_release ; objc_release
mov     x0, x19
mov     x1, x20
bl      imp___stubs__objc_msgSend ; objc_msgSend
mov     x29, x29
bl      imp___stubs__objc_retainAutoreleasedReturnValue ; objc_retainAutoreleasedReturnValue
mov     x21, x0
adrp    x8, #0x10031f000
ldr     x1, [x8, #0xbc0] ; "text",@selector(text)
bl      imp___stubs__objc_msgSend ; objc_msgSend
mov     x29, x29
```

b. NSLog() is used twice to print data on iOS Console.

c. AES256 function is used as encryption/decryption algorithm. In this example AES256Encryption is called.

d. The result of AES256 function is stored in file named encrypted.

```
mov     x20, x0
mov     x0, x22
bl      imp___stubs__objc_release ; objc_release
mov     x0, x21
bl      imp___stubs__objc_release ; objc_release
str     x20, [sp, #0x50 + var_50]
adrp    x0, #0x1002cc000 ; 0x1002cce38@PAGE
add     x0, x0, #0xe38 ; 0x1002cce38@PAGEOFF, @"plain data is %@"
bl      imp___stubs__NSLog ; NSLog
ldr     x2, [x19, x23]
adrp    x8, #0x100321000 ; &@selector(resume)
ldr     x1, [x8, #0x518] ; "dataUsingAES256EncryptionWithKey:",@selector(dataUsingAES256EncryptionWithKey:)
mov     x0, x20
bl      imp___stubs__objc_msgSend ; objc_msgSend
mov     x29, x29
bl      imp___stubs__objc_retainAutoreleasedReturnValue ; objc_retainAutoreleasedReturnValue
mov     x21, x0
str     x21, [sp, #0x50 + var_50]
adrp    x0, #0x1002cc000 ; 0x1002cce58@PAGE
add     x0, x0, #0xe58 ; 0x1002cce58@PAGEOFF, @"encryted data is %@"
bl      imp___stubs__NSLog ; NSLog
adrp    x8, #0x100320000 ; &@selector(numberWithBool:)
ldr     x1, [x8, #0xf00] ; "getPathForFilename:",@selector(getPathForFilename:)
adrp    x2, #0x1002cc000 ; 0x1002cce78@PAGE
add     x2, x2, #0xe78 ; 0x1002cce78@PAGEOFF, @"encrypted"
mov     x0, x19
bl      imp___stubs__objc_msgSend ; objc_msgSend
mov     x29, x29
bl      imp___stubs__objc_retainAutoreleasedReturnValue ; objc_retainAutoreleasedReturnValue
mov     x19, x0
adrp    x8, #0x100323000 ; &@selector(pageTable_insertForPageKeyStatement)
ldr     x1, [x8, #0x4[illegible]0] ; "writeToFile:atomically:",@selector(writeToFile:atomically:)
orr     w3, wzr, #0x1
mov     x0, x21
mov     x2, x19
bl      imp___stubs__objc_msgSend ; objc_msgSend
mov     x0, x19
bl      imp___stubs__objc_release ; objc_release
mov     x0, x21
bl      imp___stubs__objc_release ; objc_release
```

Step 5: Let’s change the view into Pseudo Code of Procedure (Modes in menu bar). Now the above disassembled code is easy to understand.

```
/* @class BrokenCryptographyExerciseViewController */
-(void)viewDidLoad {
    [[&saved_fp - 0x20 super] viewDidLoad];
    objc_storeStrong((int64_t *)&self->encryptionKey, @"myencrytionkey");
    r0 = [self textField];
    r0 = [r0 retain];
    [r0 setText:@"b@nkP@ssword123"];
    [r0 release];
    r0 = [self textField];
    r0 = [r0 retain];
    var_48 = r0;
    r0 = [r0 text];
    r0 = [r0 retain];
    var_28 = [[r0 dataUsingEncoding:0x4] retain];
    [r0 release];
    [var_48 release];
    stack[-112] = var_28;
    NSLog(@"plain data is %@", @selector(dataUsingEncoding:));
    var_30 = [[var_28 dataUsingAES256EncryptionWithKey:self->encryptionKey] retain];
    stack[-112] = var_30;
    NSLog(@"encryted data is %@", @selector(dataUsingAES256EncryptionWithKey:));
    [var_30 writeToFile:[[self getPathForFilename:@"encrypted"] retain] atomically:0x1];
    objc_storeStrong(&var_38, 0x0);
    objc_storeStrong(&var_30, 0x0);
    objc_storeStrong(&saved_fp - 0x28, 0x0);
    return;
}
```

Step 6: By looking into iOS Console we can find result printed by NSLog(). To open iOS Console you have to run XCode and then open Window -> Devices and Simulators. You can also use Console application from Mac OS X.

```
16:36:00.902697    iGoat    plain data is <62406e6b 50407373 776f7264 313233>
16:36:00.903170    iGoat    encryted data is <ec993f14 664d0efe 2d396480 6d39f9e0>
```

You can download all data of the OWASP iGoat application from Devices and Simulators and then see the content of the encrypted file. On jailbroken device you can log in by ssh then also display the content of the encrypted file.

```
Documents root# hexdump -C encrypted
00000000  ec 99 3f 14 66 4d 0e fe  2d 39 64 80 6d 39 f9 e0  |..?.fM..-9d.m9..|
```

Step 7: The extra task is to develop own implementation of AES256 function. The password must have the size equal 32 and the IV must be zero.

```
prevenity@prevenity-VirtualBox:~/Downloads/tmp$ cat aes.py
from Crypto.Cipher import AES
from pkcs7 import PKCS7Encoder

password = str("b@nkP@ssword123").encode('utf-8')
pad2 = 18 * '\x00'

key = str("myencrytionkey")
keyn = key + pad2

print(len(password))
print(len(keyn))

iv = 16 * '\x00'
encoder = PKCS7Encoder()
ciper = AES.new(keyn,AES.MODE_CBC, iv)
pad_text = encoder.encode(password)
msg = ciper.encrypt(pad_text)

print(msg)
```

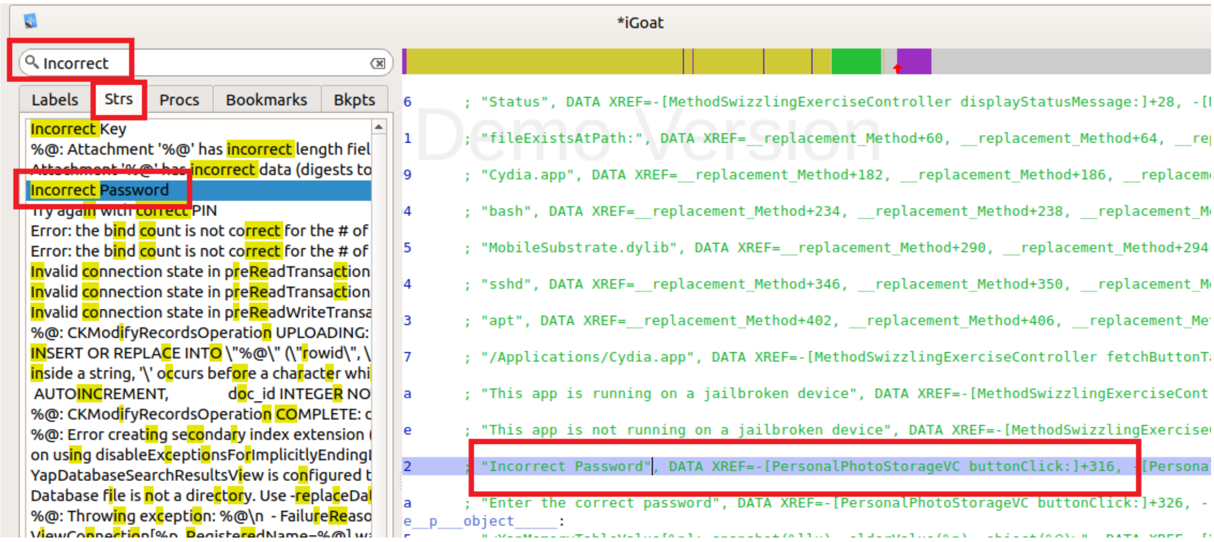
As we can see the result is the same. We were able to generate the same value.

```
prevenity@prevenity-VirtualBox:~/Downloads/tmp$ python aes.py | hexdump -C
00000000  31 35 0a 33 32 0a ec 99  3f 14 66 4d 0e fe 2d 39  |15.32...?.fM..-9|
00000010  64 80 6d 39 f9 e0 0a                              |d.m9...|
```

Exercise #2

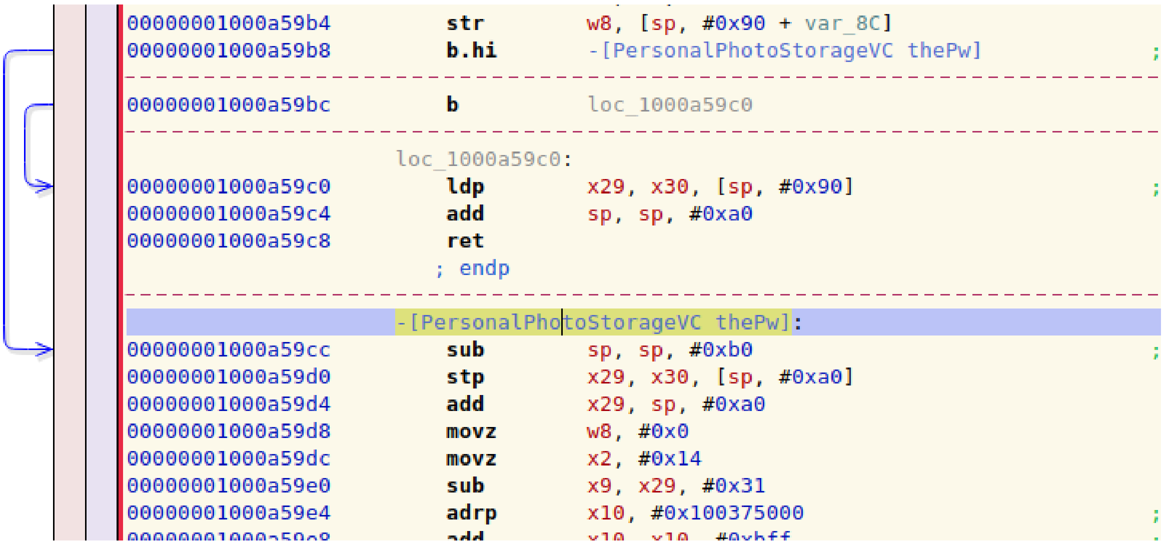
Exercise Name: **Runtime Analysis -> Personal Photo Storage**
Question: What is the right password to local storage?

Step 1: As in previous example we can start from finding string displayed after providing wrong password. In this case it is: “Incorrect Password”



Step 2: All Cross references point to class [PersonalPhotoStorageVC] and method buttonClick.

Step 3: Just after the method buttonClick we found thePW method. (**Note:** the Hopper disassembler did not display thePW method in the Procs tab, IDA Pro works fine) .



Step 4: Now, we see that XOR encryption is implemented in the PW method. The one key is 1234567890 and the second is ^BVZFSDYTU.

```
if (r8 > 0x0) {
        r31 = r31 - 0xb0;
        var_A0 = r29;
        stack[-168] = r30;
        r29 = &var_A0;
        r16 = *___stack_chk_guard;
        var_8 = **___stack_chk_guard;
        memcpy(r29 - 0x12, "^BVZFSDYTU", 0xa);
        memcpy(r29 - 0x1d, "1234567890", 0xb);
        memset(r29 - 0x31, zero_extend_64(0x0), 0x14);
        var_4C = 0x0;
        while (sign_extend_64(var_4C) < 0xa) {
                r14 = 0xa;
                r8 = r29 - 0x31;
                r9 = r29 - 0x1d;
                r10 = 0xb;
                r13 = sign_extend_64(*(int8_t *)((r29 - 0x12) + sign_extend_64(var_4C)));
                r11 = sign_extend_64(var_4C);
                asm { udiv       x14, x11, x10 };
                *(int8_t *)(r8 + sign_extend_64(var_4C)) = r13 ^ sign_extend_64(*(int8_t *)(r9 + (r11 - r14 * r10)));
                var_4C = var_4C + 0x1;
        }
        var_138 = [NSString stringWithUTF8String:r29 - 0x31];
        if (**___stack_chk_guard != var_8) {
                __stack_chk_fail();
        }
}
```

Step 5: The final step is to decode the password.

```
prevenity@prevenity-VirtualBox:~/Downloads/tmp$ cat owasp_xor.py
from itertools import izip, cycle

key = "^BVZFSDYTU"
password = "1234567890"

encrypted = ''.join(chr(ord(x) ^ ord(y)) for (x,y) in izip(password, cycle(key)))

print(encrypted)
```

Exercise #3

Exercise Name: **Runtime Analysis -> Runtime Brute Force Attack**
Question: Can you find out the correct 4 digits PIN?

Step 1: Let's start from tracing all cryptography functions in the iGoat application by using Frida.

```
$ frida-trace -U iGoat -I libcommonCrypto.dylib
…
Started tracing 208 functions. Press Ctrl+C to stop.
        /* TID 0x403 */
 12925 ms  CC_SHA256()
 12925 ms     | CCDigest()
```

Step 2: Above Frida output displayed only SHA256 calls. Now we can modify CC_SHA256.js script to print arguments and the result of sha256 function or we can create the new js file.

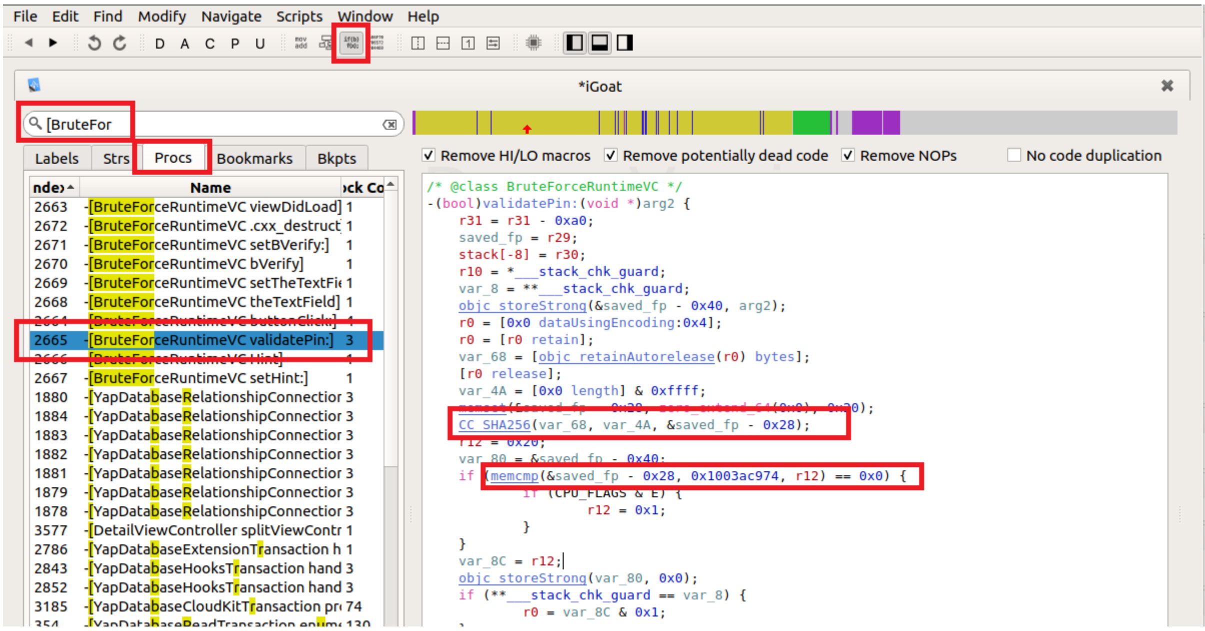
```
Interceptor.attach(Module.findExportByName('libcommonCrypto.dylib', 'CC_SHA256'),{
 onEnter: function(args) {
 var data = hexdump(args[0], {
  length: args[1].toInt32()
  });
```

```
  console.log("\n SHA256(" + data + ")\n");
  console.log("Backtrace:\n\t");
  console.log(Thread.backtrace(this.context,
Backtracer.ACCURATE).map(DebugSymbol.fromAddress).join("\n"));

}
 ,onLeave: function (retval) {
 console.log("\n sha256 result: " + hexdump(retval, {
   length: 32 }) + " \n");
 }
});
```

Step 3: Now we will start the Frida and the iGoat applications once again. We will confirm that provided PIN is hashed by using the SHA256 function. But I also wanted to identify classes and methods which calls SHA256 to calculate hash from 4 digit PIN value. Backtrace function displayed the class BruteForceRuntimeVC with the method validatePin [BruteForceRuntimeVC validatePin:].

```
$ frida -U -l example_crypt.js iGoat
     ____
    / _  |   Frida 11.0.11 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/

[iPhone::iGoat]->
 SHA256(         0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  31 31 31 31                                      1111)

Backtrace:

0x1001c07d4 iGoat!-[BruteForceRuntimeVC validatePin:]
0x1001c04d0 iGoat!-[BruteForceRuntimeVC buttonClick:]
0x1987ffd30 UIKit!-[UIApplication sendAction:to:from:forEvent:]
…

 sha256 result:           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  0f fe 1a bd 1a 08 21 53 53 c2 33 d6 e0 09 61 3e  ......!SS.3...a>
00000010  95 ee c4 25 38 32 a7 61 af 28 ff 37 ac 5a 15 0c  ...%82.a.(.7.Z..
```

Step 4: It is time to open the Hopper disassembler and will try to find this method in disassembled code of the OWASP iGoat application. It is recommended to use top bar menu button to generate Pseudo Code.



We can notice that after calling SHA256 function, the result (stored at &saved_fp – 0x28) is compared by using the function memcmp(). The second attribute is at address 0x1003ac974. Also by changing view to Assembly we can easily identify reference to data - at address 0x1003ac974.

```
00000001001007d0    bl      imp___stubs__CC_SHA256          ; CC_SHA256
00000001001007d4    movz    x8, #0x0
00000001001007d8    sub     x9, x29, #0x40
00000001001007dc    sub     x10, x29, #0x28
00000001001007e0    adrp    x1, #0x1003ac000                ; 0x1003ac974@PAGE
00000001001007e4    add     x1, x1, #0x974                  ; 0x1003ac974@PAGEOFF, _validatePin:.reference
00000001001007e8    orr     w12, wzr, #0x20
00000001001007ec    mov     x2, x12
00000001001007f0    str     x0, [sp, #0x90 + var_78]
00000001001007f4    mov     x0, x10
00000001001007f8    str     x9, [sp, #0x90 + var_80]
00000001001007fc    str     x8, [sp, #0x90 + var_88]
0000000100100800    bl      imp___stubs__memcmp             ; memcmp
```

Finally, we can display table with binary values – this is the SHA256 value from the correct 4 digit PIN.

```
                        _validatePin:.reference:
00000001003ac974        db      0x02 ; '.'
00000001003ac975        db      0x52 ; 'R'
00000001003ac976        db      0xb0 ; '.'
00000001003ac977        db      0x81 ; '.'
00000001003ac978        db      0xbd ; '.'
00000001003ac979        db      0xa7 ; '.'
00000001003ac97a        db      0x0b ; '.'
00000001003ac97b        db      0x47 ; 'G'
00000001003ac97c        db      0x8f ; '.'
00000001003ac97d        db      0x01 ; '.'
00000001003ac97e        db      0x31 ; '1'
00000001003ac97f        db      0xb3 ; '.'
00000001003ac980        db      0x10 ; '.'
00000001003ac981        db      0xa9 ; '.'
00000001003ac982        db      0x3c ; '<'
00000001003ac983        db      0xb8 ; '.'
00000001003ac984        db      0xd7 ; '.'
```

Step 5. The very last step is to write some code to brute force the valid PIN.

```
preventity@preventity-VirtualBox:~/Downloads/tmp$ cat python.py
from itertools import product
import hashlib
hash1 = '0252b081bda70b478f0131b310a93cb8d79086d785fb4ae392a8c5ffc3ddc5fe'
print(hash1)
for i in range(10000):
   hash2 = hashlib.sha256(str(i).encode('utf-8')).hexdigest()
   if hash1 == hash2:
    print(hash2)
    print(str(i))
```

End of 3nd exercise.

References

[1] https://www.owasp.org/index.php/OWASP_iGoat_Tool_Project

[2] https://github.com/owasp/igoat

[3] https://www.hopperapp.com/

[4] https://frida.re/

[5] http://armconverter.com/