Bypass OpenSSL Certificate Pinning on iOS

Using Binary Patching and In-Memory Hooking

Daniel Mayer (daniel@matasano.com)

January 7, 2015

Abstract:
When mobile applications communicate with an API or web service, this should generally happen via TLS/SSL (e.g., HTTPS). In order to verify the identity of the server and to prevent man-in-the-middle attacks, TLS relies on certificates which prove the identity of the web server. Browsers and mobile operating systems come preconfigured with a list of trusted Certificate Authorities (CAs). Since any of the CAs may issue a certificate for any hostname/server, security-conscious applications should "pin" the expected server certificate in the application, i.e., not accept any certificate but the one issued by the known-good CA which the application developer uses.

From a penetration testing perspective, this may cause practical problems since it is difficult to intercept the communication of an application that makes use of this technique. Without pinning, interception typically involves adding the TLS certificate of an intercepting proxy (such as Burp) to the certificate store of the target operating system. However, when the app uses certificate pinning, this store is often ignored. On iOS, when the app uses standard iOS APIs, the iOS SSL Kill Switch, developed by Matasano's sister company iSEC Partners, can be used to bypass pinning and force the application to accept any certificate presented by the server or proxy. The Kill Switch uses Cydia Substrate which hooks the iOS functions used for certificate validation and modifies them so that they accept any certificate. It becomes more complicated when the app uses the OpenSSL library instead of the native iOS frameworks since they are not affected by the Kill Switch's hooking.

There is more than one way of bypassing OpenSSL-based certificate pinning and, in this paper we detail two of these methods: binary patching and Cydia Substrate.

1 the scenario: mock-up app on github

To make this paper easier to follow, we created a mock-up iOS application that uses OpenSSL and pins certificates in a particular way. There are many different ways for implementing pinning and the solutions presented in this paper are transferable to most of them.

The mock-up app simply connects to https://www.example.org and performs a `GET` request for `/`. The result is then displayed in a simple text view. Some debug information is output via `NSLog` to make debugging easier. The relevant code is in the `ViewController.mm` file.

The repository contains two XCode projects, one which builds ARMv7 and one that builds ARMv8 executables. In addition, the `binaries` folder includes example binaries to illustrate the patching performed in this whitepaper.

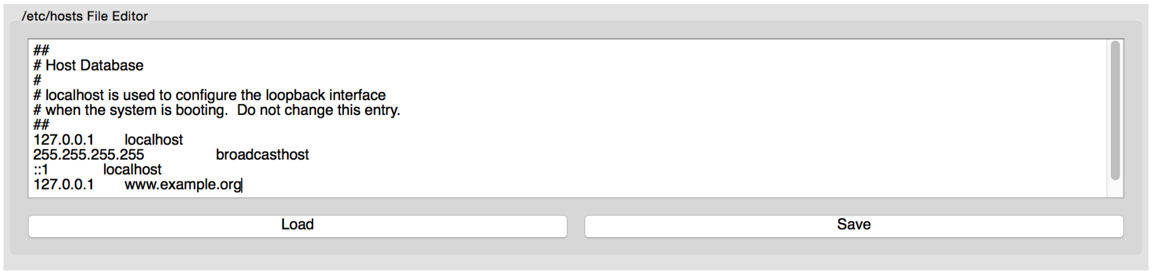
Note, in order to perform the steps and modifications described in this whitepaper yourself, you must have a jailbroken iDevice. The main reason is that we will require root-level device access as well as disabled application signature checking.

Before we dive into the pinning bypass, in the next section, we first describe how to position ourselves between the app and `www.example.org`.

2 redirecting app traffic to burp

The challenges when an app uses OpenSSL instead of native iOS functions begin before the pinning gets in the way. In order to intercept application traffic, it is usually sufficient to set the iOS system proxy such that it points at the intercepting proxy. However, if the app uses C(++) code to open a TCP socket via OpenSSL, the OS is not queried for the iOS proxy settings.

One common way around this is to modify the hostname to IP address mapping used by the application. One could stand up a custom DNS server and configure the device to use it—effectively making the app connect to an IP address of our choosing. An easier way to achieve the same result is to modify the device's `/etc/hosts` file. Using my tool, idb, this is can easily be accomplished under the "Tools" tab:



```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1       localhost
255.255.255.255         broadcasthost
::1             localhost
127.0.0.1       www.example.org
```

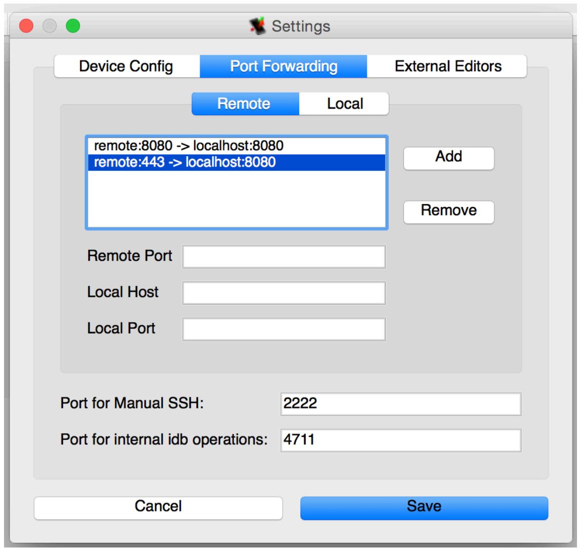
Here we point the server `www.example.org` to resolve to the loopback address `127.0.0.1`:

LISTING 1: HOSTS FILE ENTRY

```
127.0.0.1 www.example.org
```

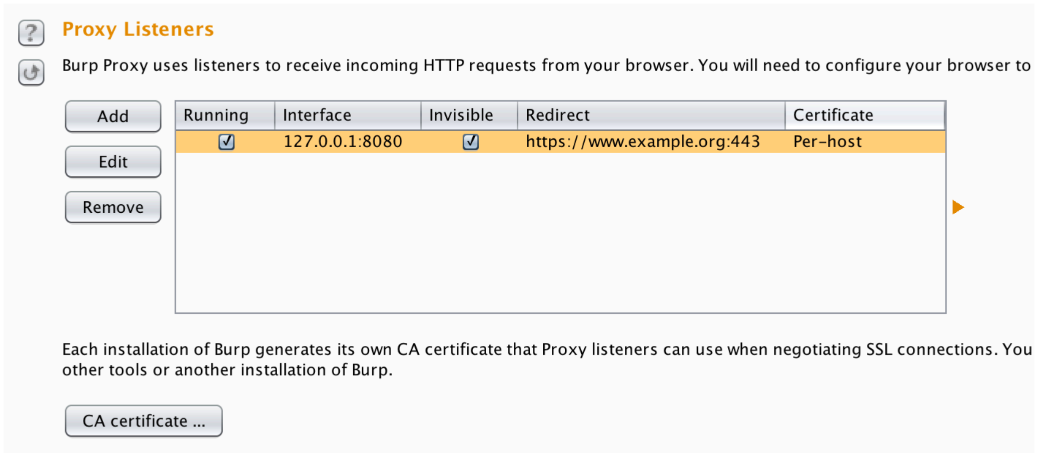
We can then use SSH forwarding (e.g., in idb) to forward traffic from port 443 on the iDevice to port 8080 on our workstation, where Burp is listening and transparently proxying requests to example.org.

Matasano Security Research



remote:8080 -> localhost:8080
remote:443 -> localhost:8080

Remote Port	
Local Host	
Local Port	
Port for Manual SSH:	2222
Port for internal idb operations:	4711



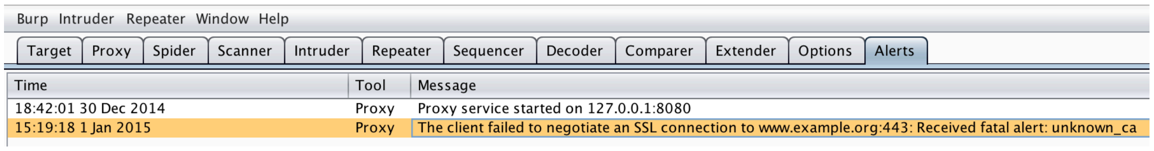
Proxy Listeners

Burp Proxy uses listeners to receive incoming HTTP requests from your browser. You will need to configure your browser to

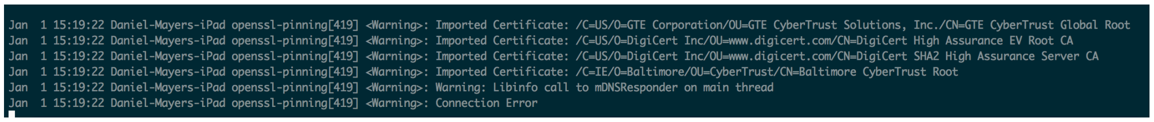
Running	Interface	Invisible	Redirect	Certificate
✓	127.0.0.1:8080	✓	https://www.example.org:443	Per-host

Each installation of Burp generates its own CA certificate that Proxy listeners can use when negotiating SSL connections. You other tools or another installation of Burp.

The application now tries to connect to Burp, but fails due to certificate pinning. Note that it may be tricky to figure out why the connection fails. Check the 'Alerts' tab in Burp, or if you are lucky there may be some app logging that discloses a certificate validation failure.



Time	Tool	Message
18:42:01 30 Dec 2014	Proxy	Proxy service started on 127.0.0.1:8080
15:19:18 1 Jan 2015	Proxy	The client failed to negotiate an SSL connection to www.example.org:443: Received fatal alert: unknown_ca



```
Jan  1 15:19:22 Daniel-Mayers-iPad openssl-pinning[419] <Warning>: Imported Certificate: /C=US/O=GTE Corporation/OU=GTE CyberTrust Solutions, Inc./CN=GTE CyberTrust Global Root
Jan  1 15:19:22 Daniel-Mayers-iPad openssl-pinning[419] <Warning>: Imported Certificate: /C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert High Assurance EV Root CA
Jan  1 15:19:22 Daniel-Mayers-iPad openssl-pinning[419] <Warning>: Imported Certificate: /C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High Assurance Server CA
Jan  1 15:19:22 Daniel-Mayers-iPad openssl-pinning[419] <Warning>: Imported Certificate: /C=IE/O=Baltimore/OU=CyberTrust/CN=Baltimore CyberTrust Root
Jan  1 15:19:22 Daniel-Mayers-iPad openssl-pinning[419] <Warning>: Warning: Libinfo call to mDNSResponder on main thread
Jan  1 15:19:22 Daniel-Mayers-iPad openssl-pinning[419] <Warning>: Connection Error
```

3 circumventing openssl certificate pinning

As mentioned above, we cannot simply use iSEC's SSL Kill Switch to turn off SSL certificate validation. One alternative approach is to hook and override the OpenSSL certificate validation functions using Cydia Substrate, which we will discuss later in this paper. But first, the next section will explore patching the application binary to bypass pinning.

3.1 ANALYZING THE PINNING MECHANISM

The main mechanism this app uses for pinning is a restricted CA list. Instead of relying on a third-party collection of certificates, the app dynamically generates an OpenSSL certificate store in-memory (`X509_STORE * cert_store = X509_STORE_new();`) and provisions it with the single known-good certificate chain using `load_cert`:

LISTING 2: FUNCTION WHICH POPULATES IN-MEMORY STORE

```
void load_cert(char pem[], X509_STORE *cert_store) {
    BIO * bio = BIO_new_mem_buf(pem, (int)strlen(pem));
    pinned_cert = PEM_read_bio_X509(bio, NULL, NULL, NULL);
    X509_STORE_add_cert(cert_store, pinned_cert);
}
```

The trusted CA certificates are hard-coded in the app source code (in PEM format). For example:

LISTING 3: PEM CERTIFICATES PINNED IN BINARY

```
/**
 * Adding the root CA and the intermediary CA certificates to the trust store
 * example.org has 2 intermediaries in addition to the root CA cert.
 * We need all of them in order to validate the entire chain.
 */
void setup_pinned_ca_certs() {

    char root_ca_cert[] = "-----BEGIN CERTIFICATE-----\n"
    "MIICWjCCAcMCAgGlMA0GCSqGSIb3DQEBBAUAMHUxCzAJBgNVBAYTAlVTMRgwFgYD\n"
    "VQQKEw9HVEUgQ29ycG9yYXRpb24xJzAlBgNVBAsTHkdURSBDeWJlclRydXN0IFNv\n"
    "bHV0aW9ucywgSW5jLjEjMCEGA1UEAxMaR1RFIEN5YmVyVHJ1c3QgR2xvYmFsIFJv\n"
    "b3QwHhcNOTgwODEzMDAyOTAwWhcNMTgwODEzMjM1OTAwWjB1MQswCQYDVQQGEwJV\n"
    "UzEYMBYGA1UEChMPR1RFIENvcnBvcmF0aW9uMScwJQYDVQQLEx5HVEUgQ3liZXJU\n"
    "cnVzdCBTb2x1dGlvbnMsIEluYy4xIzAhBgNVBAMTGkdURSBDeWJlclRydXN0IEds\n"
    "b2JhbCBSb290MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCVD6C28FCc6HrH\n"
    "iM3dFw4usJTQGz0O9pTAipTHBsiQl8i4ZBp6fmw8U+E3KHNgf7KXUwefU/ltWJTS\n"
    "r41tiGeA5u2ylc9yMcqlHHK6XALnZELn+aks1joNrI1CqiQBOeacPwGFVw1Yh0X4\n"
    "04Wqk2kmhXBIgD8SFcd5tB8FLztimQIDAQABMA0GCSqGSIb3DQEBBAUAA4GBAG3r\n"
    "GwnpXtlR22ciYaQqPEh346B8pt5zohQDhT37qw4wxYMWM4ETCJ57NE7fQMh017l9\n"
    "3PR2VX2bY1QY6fDq81yx2YtCHrnAlU66+tXifPVoYb+O7AWXX1uw16OFNMQkpw0P\n"
    "lZPvy5TYnh+dXIVtx6quTx8itc2VrbqnzPmrC3p/\n"
    "-----END CERTIFICATE-----";

    load_cert(root_ca_cert, cert_store);
```

In general, it is common to store CA certificates in the file system and it would be natural to do the same for pinned certificates in mobile apps. One downside of that approach is that the certificates can easily be swapped out on a jailbroken device. In contrast, certificates that live in the binary are more difficult to exchange, since one has to modify the binary itself to do so. As we will see, neither will stop a dedicated reverse-engineer from bypassing this kind of mechanism.

It is important to note that access to the application source code is not required in order to apply the techniques described in this paper. We just include it here in order to better illustrate the used techniques. Once the relevant functions are known, it is possible to reverse engineer the pinning mechanism from the binary alone.

3.2 BYPASSING PINNING USING BINARY PATCHING

Given this setup, we can either exchange the certificate in the binary or disable the certificate validation another way. Exchanging the certificates is challenging since different certificates have different lengths and the space in the binary where the original certificates are located may not be sufficient. In addition, large edits in binaries can be error-prone. So let us instead look at the code that enables certificate validation in OpenSSL:

LISTING 4: ENABLING CERTIFICATE VALIDATION

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
```

A quick look at the documentation for `SSL_CTX_set_verify` tells us that we need to turn `SSL_VERIFY_PEER` into `SSL_VERIFY_NONE` to disable certificate validation. Both of these values are constants defined in ssl.h:

LISTING 5: DEFINITION OF VALIDATION FLAGS

```
#define SSL_VERIFY_NONE                 0x00
#define SSL_VERIFY_PEER                 0x01
```

This means we will need to find the call to `SSL_CTX_set_verify` in the app binary, and change the second argument from `0x01` to `0x00`. It is important to note that such modifications will break app signing so you must use a jailbroken device which disables signature verification.

Pull the app binary from the device (or the binaries folder on Github) and disassemble it with `otool` which comes pre-installed on OS X and can be downloaded and compiled for Linux. (Note that you may need to decrypt the app binary if it comes from an app store. idb is able to do this by internally using dumpdecrypted.)

LISTING 6: DISASSEMBLY USING OTOOL

```
 otool -Vjt openssl-pinning
```

Note that if you happen to have Hopper or IDA this may be much easier, but let us continue down the manual path. All modern iOS devices (Apple A7 or better) use the ARMv8-A architecture which has full 64-bit support so we will see a 64-bit address space in our binary. For the sake of educational simplicity let us start by looking at a binary compiled for ARMv7 first and at the 64-bit version afterwards.

3.2.1 ARMv7 Binary

Searching for `SSL_CTX_set_verify` in the disassembly gives us:

LISTING 7: ASSEMBLY CALLING SSL_CTX_SET_VERIFY

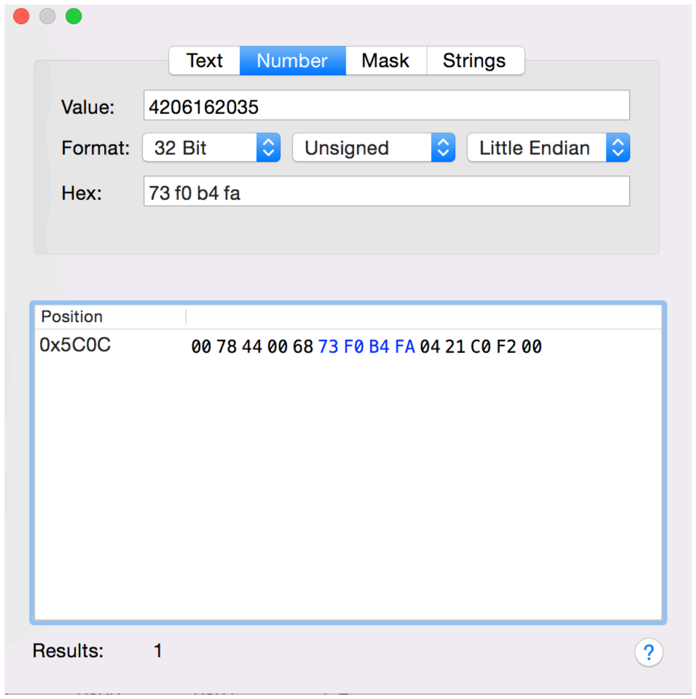
```
00009bf4           2101     movs    r1, #0x1
00009bf6       f2c00100     movt    r1, #0x0
00009bfa           2200     movs    r2, #0x0
00009bfc       f2c00200     movt    r2, #0x0
00009c00       f24c60e8     movw    r0, #0xc6e8
00009c04       f2c0000b     movt    r0, #0xb
00009c08           4478     add     r0, pc
00009c0a           6800     ldr     r0, [r0]
00009c0c       f073fab4     bl      _SSL_CTX_set_verify
```

The important thing to note here are the lines before the branch and link (`bl _SSL_CTX_set_verify`) in which the arguments are set up. ARM assembly normally passes function arguments in registers. `SSL_CTX_set_verify` takes three arguments which are passed via `r0`, `r1`, and `r2`. These registers are set here using 16-bit instructions (ARM thumb mode). Because of that, the registers are set using two instructions `movs` and `movt` where `movs` writes the lower half-word and `movt` the upper half-word of the register. Since we are only concerned with modifying the lowest bit, we can ignore the `movt` instruction.

Register `r0` will point to the OpenSSL `ctx` and `r3` should be `NULL` which matches the assignments in the disassembly. Then `r1` holds the second argument to `SSL_CTX_set_verify` which is `#0x1` (=`SSL_VERIFY_PEER`). Recall that `SSL_VERIFY_NONE = 0` so changing the instruction at `00009bf4` from 2101 to 2100 gives us `movs r1, #0x0` and should disable cert validation (MOVS documentation).

Now finding that instruction in the binary is a bit painful since the addresses used by otool are memory locations and not the offset in the file on disk. I found it easiest to search for a sequence of opcodes/instructions in a hex editor (I used Synalyze It! in the past). Another aside: byte order can be annoying since in little endian e.g., `f073fab4` will be `73f0b4fa` in the binary. Searching for `73f0b4fa`:

Matasano Security Research



Value: 4206162035

Format: 32 Bit | Unsigned | Little Endian

Hex: 73 f0 b4 fa

```
Position
0x5C0C        00 78 44 00 68 73 F0 B4 FA 04 21 C0 F2 00
```

Results: 1

Moving backwards in the binary it is easy to spot the bytes to be modified:

```
0x05BE0 73 F0 FA F8 4C F2 04 71 C0 F2 0B 01 79 44 08 60 FF F7 44 FF 00 21 C0 F2 00 01 01 22 C0 F2 00 02
0x05C00 4C F2 E8 60 C0 F2 0B 00 78 44 00 68 73 F0 B4 FA 04 21 C0 F2 00 01 4C F2 D2 60 C0 F2 0B 00 78 44
```

After changing the byte you can use `otool` again to ensure the patching worked as expected:

LISTING 8: PATCHED ASSEMBLY SETTING SSL_VERIFY_NONE

```
00009bf4        2100        movs     r1, #0x0
```

After re-uploading the binary to the app folder on the device, it will fail again with a different certificate validation error. Turns out this particular app is rather thorough and validates several different aspects of the received cert manually (beyond relying on the OpenSSL feature patched above). So back to patching the relevant code. In `bool verify_certificate()` there is sequence of 4 if constructs all looking similar to this:

Matasano Security Research

LISTING 9: SECONDARY CERTIFICATE VALIDATION

```
// First let us verify this is a cert in our trust store.
if(SSL_get_verify_result(ssl)!=X509_V_OK)
{
    NSLog(@"Certificate doesn't verify. Error: %s",
    X509_verify_cert_error_string(SSL_get_verify_result(ssl)));
    return false;
}
```

This code explicitly verifies the certificate against the trust store. This is one of the checks that OpenSSL already performs internally when peer verification is enabled, making this a redundant check. But the developer of this app decided to verify it one more time explicitly. Searching for `SSL_get_verify_result` in the assembly gives us something like

LISTING 10: ASSEMBLY CALLING SSL_GET_VERIFY_RESULT

```
00009e2a        f073ffdf        bl      _SSL_get_verify_result
;[ Omitting a bunch of instructions for the NSLog line ]
00009eb0        f0a5eff2        blx     0xafe98 @ symbol stub for: _NSLog
00009eb4            2000        movs    r0, #0x0
00009eb6        f2c00000        movt    r0, #0x0
00009eba        f0000001        and     r0, r0, #0x1
00009ebe        f88d0068        strb.w  r0, [sp, #104]
00009ec2            e139        b       0xa138
```

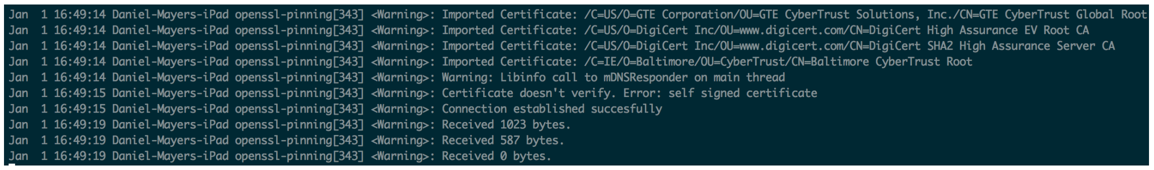
Another thing to know about ARM is that there is no RET instruction. The return value is typically passed in r0 and then a branching (`b`) instructions jumps to either the return address (from a register or stack) or in this case a hard-coded address `0xa138`. So `r0` is set to `#0x0` (false) here. In order to return true instead and trick the logic into accepting the certificate, we just change `r0` to `#0x1` which corresponds to an instruction of `2001`. We are again in thumb mode so we leave the `movt` instruction intact. (Hint: search for `8d f8 68 00 39 e1`):

```
0x05EA0 4F F6 FF 72 CF F6 FF 72 1F 92 13 90 08 46 13 99 A5 F0 F2 EF 00 20 C0 F2 00 00 00 F0 01 00 8D F8
0x05EC0 68 00 39 E1 19 98 4F F0 FF 31 1F 91 12 91 51 F0 6D F8 12 99 1F 91 0D 21 2B AA 4F F4 80 73 11 92
```

LISTING 11: PATCHED RETURN VALUE

```
00009eb4            2001        movs    r0, #0x1
```

This allows us to return early, skip all the other annoying validation functions, and our app is talking to Burp!



```
Jan  1 16:49:14 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Imported Certificate: /C=US/O=GTE Corporation/OU=GTE CyberTrust Solutions, Inc./CN=GTE CyberTrust Global Root
Jan  1 16:49:14 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Imported Certificate: /C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert High Assurance EV Root CA
Jan  1 16:49:14 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Imported Certificate: /C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert SHA2 High Assurance Server CA
Jan  1 16:49:14 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Imported Certificate: /C=IE/O=Baltimore/OU=CyberTrust/CN=Baltimore CyberTrust Root
Jan  1 16:49:14 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Warning: Libinfo call to mDNSResponder on main thread
Jan  1 16:49:15 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Certificate doesn't verify. Error: self signed certificate
Jan  1 16:49:15 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Connection established succesfully
Jan  1 16:49:19 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Received 1023 bytes.
Jan  1 16:49:19 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Received 587 bytes.
Jan  1 16:49:19 Daniel-Mayers-iPad openssl-pinning[343] <Warning>: Received 0 bytes.
```

Now that we have the basics down using ARMv7, let us look how the same process looks like for ARMv8 (ARM64).

3.2.2 ARMv8 (ARM64) binary

Interestingly, the ARMv8 assembly looks very different, which makes it slightly more challenging to work with in our case. Lets look at the disassembly and search for `SSL_CTX_set_verify` as before:

LISTING 12: ASSEMBLY CALLING SSL_CTX_SET_VERIFY

```
0000000100004f84        320003e1                orr     w1, wzr, #0x1
0000000100004f88        d2800002                movz    x2, #0
0000000100004f8c        d00007a8                adrp    x8, 246 ; 0x100005000
0000000100004f90        912b4100                add     x0, x8, #2768
0000000100004f94        f9400000                ldr     x0, [x0]
0000000100004f98        9402620b                bl      _SSL_CTX_set_verify
```

Again, the important part is the setup of the registers. The confusing part here is that registers are not named in a straightforward manner: `x` registers are the default general purpose registers and `w` registers are 32-bit sub registers (For those interested, Apple does not seem to fully follow the ARM specification for function calls; see Apple's ARM64 function call conventions).

So, here `x0` is the first argument, `w1` the second, and `x2` the third. Meaning `x0` is the pointer to the OpenSSL `ctx`, `w1` is our target value for the certificate validation flag, and `x3` should just point to `NULL`.

Lets focus on `w1`: Register `wzr` always returns `#0x0`. Then `orr` performs a bitwise OR of `#0x0` with the constant `#0x1` and the result is stored in `w1`. That means `w1` will be set to `#0x1` (=`SSL_VERIFY_PEER`).

Looking at the machine code `320003e1` for this line it is not entirely clear to me how to modify this into `orr w1, wzr, #0x0` (in fact Section 5.3.2 in the ARM reference manual suggests it is not possible). So let us modify the instruction altogether. XORing any register with itself will yield `#0x0` so `eor w1, w1, w1` should store `#0x0` into `w1`. But we must determine which opcode that corresponds to. Thankfully XCode on OS X ships with llvm-gcc (and similar tools are available for Linux) which can assemble the instruction

Matasano Security Research

LISTING 13: ASSEMBLY SETTING W1 TO 0X00

```
EOR w1, w1, w1
```

by running: `llvm-gcc -arch arm64 -c eor.s -o eor.out`. The result gives us the op codes for our desired instruction:

LISTING 14: ASSEMBLED BINARY SETTING W1 TO 0X00

```
eor.out:
(__TEXT,__text) section
0000000000000000    4a010021        eor w1, w1, w1
```

We can now use this and replace the instruction at 0000000100004f84 (320003e1) with 4a010021 just as we did for ARMv7 binaries. When searching for the location in the binary, it is important to remember that the instruction word size on this architecture is different. That means that the required little endian encoding of 9402620b (the call to `_SSL_CTX_set_verify`) is 0b620294. So after searching for that in the binary, we find the location that needs patching:

```
0x004F80 9F FF FF 97 E1 03 00 32 02 00 80 D2 A8 07 00 D0 00 41 2B 91 00 00 40 F9 0B 62 02 94 E1 03 1E 32
0x004FA0 A8 07 00 D0 00 41 2B 91 00 00 40 F9 09 62 02 94 A8 07 00 D0 00 41 2B 91 00 00 40 F9 F9 7B 02 94
```

The otool output after performing this modification will look like this:

LISTING 15: RESULTING BINARY SETTING W1 TO 0X00

```
0000000100004f84        4a010021                    eor     w1, w1, w1
```

By running the resulting binary, we will see again that the second certificate verification fails. So let us patch that one as well. The relevant section calling `_SSL_get_verify_result` looks like this:

LISTING 16: ASSEMBLY HANDLING FAILED VALIDATION

```
00000001000051d0        9402658a                    bl      _SSL_get_verify_result
;[ Omitting a bunch of instructions for the NSLog line ]
0000000100005204        94028107                    bl      0x1000a5620 ; stub for: _NSLog
0000000100005208        52800009                    movz    w9, #0
000000010000520c        12000129                    and     w9, w9, #0x1
0000000100005210        39029fe9                    strb    w9, [sp, #167]
0000000100005214        140000af                    b       0x1000054d0
```

This piece of disassembly should look familiar from above. Presumably changing `movz w9, #0` to `movz w9, #1` like we did before should be sufficient. Lets use again our assembler to see how the machine code looks for that:

LISTING 17: ASSEMBLED BINARY SETTING W9 TO 0X1

```
(__TEXT,__text) section
0000000000000000    52800029        movz    w9, #1
```

By editing the binary and modifying the instruction we get the desired result:

LISTING 18: RESULTING BINARY SETTING W9 TO 0X01

```
0000000100005208        52800029                    movz    w9, #1
```

Now all is left to replace the original binary on the device with the patched version and we have bypassed the pinning once again!

4 bypassing pinning using cycript

The tool cycript allows interactive hooking via Cydia Substrate. It is possible to achieve the same result we did with the binary patching by simply modifying the behavior of functions in memory. Let's follow the same flow as we did before.

Before we get started, we need to hook the running `openssl-pinning` process on the device and open the interactive cycript shell. For this, SSH into the iDevice and execute:

LISTING 19: HOOKING THE OPENSSL-PINNING APP

```
$ cycript -p openssl-pinning
```

The basic idea now is to find where the definition of `SSL_CTX_set_verify` is located in memory and to then override the body of that function with custom code. In cycript we can do that by using dynamic symbol lookup as follows:

LISTING 20: LOCATING SSL_CTX_SET_VERIFY IN MEMORY

```
cy# @import com.saurik.substrate.MS

cy# ssl_verify = dlsym(RTLD_DEFAULT, 'SSL_CTX_set_verify')
0x106179
```

We just need to modify the the second function argument by setting it to `0` (= `SSL_VERIFY_NONE`) and cydia substrates `HookFunction` conveniently lets us call the original implementation of `SSL_CTX_set_verify` with modified arguments:

LISTING 21: MODIFYING FUNCTION ARGUMENTS

```
cy# ssl_verify = @encode(void*(void *, int, void *))(ssl_verify)
0x106179

cy# SSL_VERIFY_NONE = 0
cy# oldf = {}
{}

cy# MS.hookFunction(ssl_verify, function(arg0, verify_cert, arg2) {
cy> var ret = (*oldf)(arg0, SSL_VERIFY_NONE, arg2);
cy> return ret;
cy> },oldf)
cy#
```

Here we first set the function signature for `ssl_verify` and then define `oldf` which will be used as a reference to the original function. Then we hook the original function and provide a new function body which simply calls `oldf` with modified arguments, setting the second argument to `0`. Without exiting cycript, if we now perform the request in the app we see the first verification passes and we are stuck at the second validation function.

We now could proceed by hooking all of the functions called in `verify_certificate`. But the faster way is to simply hook `verify_certificate` and have it always return `true`. For that, we need to know where this function is located in memory. Unfortunately, `dlsym` is unable to locate this function as is. Compared to `SSL_CTX_set_verify`, `verify_certificate` is not an exported function and it is therefore not available under that symbolic name. Non-exported functions are mangled when the Objective-C is compiled. By running `nm` on the binary, we can determine the mangled name:

LISTING 22: DETERMINING THE SYMBOL

```
$ nm openssl-pinning|grep verify_certificate
00009de0 T __Z18verify_certificatev
```

So the mangled symbol we are looking for is `__Z18verify_certificatev`. One caveat here is that the symbol output by `nm` has an additional underscore prefix so we are actually looking for `_Z18verify_certificatev`. Now that we know the symbol's name, let us look it up in the binary and re-implement it in memory so that it doesn't perform any validation and simply returns true:

LISTING 23: HOOKING VERIFY_CERTIFICATE

```
cy# verify_cert = dlsym(RTLD_DEFAULT, '_Z18verify_certificatev')
0x79de1

cy# verify_cert = @encode(bool())(verify_cert)
0x79de1

cy# oldf2 = {}
{}

cy# MS.hookFunction(verify_cert, function() {
return true;
}, oldf2)
cy#
```

After this modification, `verify_certificate` always returns `true` and the pinning is successfully bypassed.

Once familiar with cycript and its functions, this second version for bypassing pinning is significantly easier to pull off than patching the application binary. However, the binary patching technique may be more versatile for platforms where cycript and Cydia Substrate are not readily available.

5 conclusion

Certificate pinning is a helpful technique to protect against rogue CAs or to ensure that corporate proxies which intercept TLS traffic are unable to access sensitive application traffic (note that your app will break, but data is protected). From a penetration testing perspective, they can pose challenges it is certainly always possible to do so! Be it via the iOS SSL Kill Switch, manual cycript hooking, or binary patching as described above. Therefore, relying on certificate pinning to prevent an attacker from learning more about the underlying communication protocol used by the application is merely security by obscurity. This is neither the intended purpose of pinning nor is it effective.

Thanks for reading and please don't hesitate to get in touch for questions or point out mistakes/errors (Email: daniel@matasano.com, Twitter: @DanlAMayer).

ABOUT THE AUTHOR

Daniel is a senior security consultant with Matasano having experience in application and network penetration testing, mobile security, security research, and system and network administration. He also has expertise in the analysis and design of cryptographic protocols, and the detection and evaluation of timing side-channels. While working at Matasano, Daniel developed a tool called 'idb' for iOS application penetration testing and presented research at various security conferences including Black Hat USA, ShmooCon, Toorcon, SOURCE Boston, and THOTCON.

Prior to Matasano, Daniel was a researcher at the Stevens Institute of Technology where his dissertation was in the area of applied cryptography and privacy.

As a co-founder and CTO of a web hosting and web development company, Daniel has more than thirteen years of experience with Internet technologies.

Daniel holds a Ph.D. degree in Computer Science from the Stevens Institute of Technology and a Masters degree in Physics from Rutgers University.