COMPREHENSIVE REPORT

ACME CORPORATION

SECURITY ASSESSMENT 2050

FEBRUARY 28, 2050

BISHOP FOX

This engagement was performed in accordance with the Statement of Work, and the procedures were limited to those described in that agreement. The findings and recommendations resulting from the assessment are provided in the attached report. Given the time-boxed scope of this assessment and its reliance on client-provided information, the findings in this report should not be taken as a comprehensive listing of all security issues.

This report is intended solely for the information and use of Acme Corporation.

Bishop Fox Contact Information:
+1 (480) 621-8967
contact@bishopfox.com
8240 S. Kyrene Road
Suite A-113
Tempe, AZ 85284

TABLE OF CONTENTS

EXECUTIVE REPORT

Project Overview

Acme Corporation engaged Bishop Fox to assess the security of the Acme infrastructure and `www.acme.com`. The following report details the findings identified during the course of the engagement, which started on January 1, 2050.

Goals

- Perform a comprehensive architecture assessment against enterprise infrastructure to identify attack vectors against critical data assets
- Identify attack vectors for PCI information at retail locations, including point-of-sale (POS) systems
- Breach the security of the business-critical enterprise and mobile applications on the Acme network

Finding Counts
2 Critical
2 High

4 Total findings

Scope
Acme infrastructure

`www.acme.com`

Dates
01/01/2050
Kickoff

01/02/2050 – 02/17/2050
Active testing

02/28/2050
Report delivery

2,144
Extracted domain credentials

98
Decrypted application passwords

Summary of Findings

The assessment team found that Acme Corporation had significantly invested in security throughout the organization. However, a lack of cohesive security strategy allowed the team to exploit gaps in each layer of defense to access critical data, including customer credit card information, sensitive internal documentation, and other protected resources.

Security controls on technical infrastructure were incomplete and inconsistently implemented across the environment. Logging and monitoring successfully identified assessment team activities but failed to alert appropriate resources.

Bishop Fox recommends that Acme implement the strategic recommendations detailed in this report to close existing security gaps and improve internal infrastructure implementation to support security goals.

RECOMMENDATIONS

Footprint and Classify Data Exposure on the Internal Network — Identify where sensitive data resides within a network and document the corresponding location, access rights, information type, and sensitivity. This comprehensive listing will make subsequent actions to reduce the risk posed by sensitive information disclosure more attainable.

Deter, Prevent, Detect, Delay, Respond, and Recover from Unauthorized Access — Deploy monitoring systems, disinformation tactics, and baiting mechanisms to help expose attackers that have established presence within the Acme network. Give precedence to deploying hosts that run honeypots, honeytokens, tarpits, pseudoservers, canary traps, and software for the detection and logging of exploit attempts within the Acme network. Investigate incidents following the discovery of exploit and scanning activities to identify, contain, and eradicate attackers.

Plan for Compromise — Investigate all legal and procedural options prior to releasing Acme apps. For example, explicitly prohibit jailbroken devices in the terms and conditions of Acme applications.

ASSESSMENT REPORT

Enterprise Architecture Assessment

The assessment team conducted an enterprise architecture assessment with the following target in scope:

- Acme internal and external infrastructure

Identified Issues

1 INCONSISTENT AUTHENTICATION BOUNDARIES

CRITICAL

Definition

Authentication boundaries within Acme network are implemented on an individual application basis, leading to inconsistent authentication controls to critical data. This allows attackers to bypass security controls that should be in place to defend critical data.

Details

The assessment team identified applications that had access to critical data assets but were not protected by strong security controls. The authentication boundary was implemented as shown in the following figure:

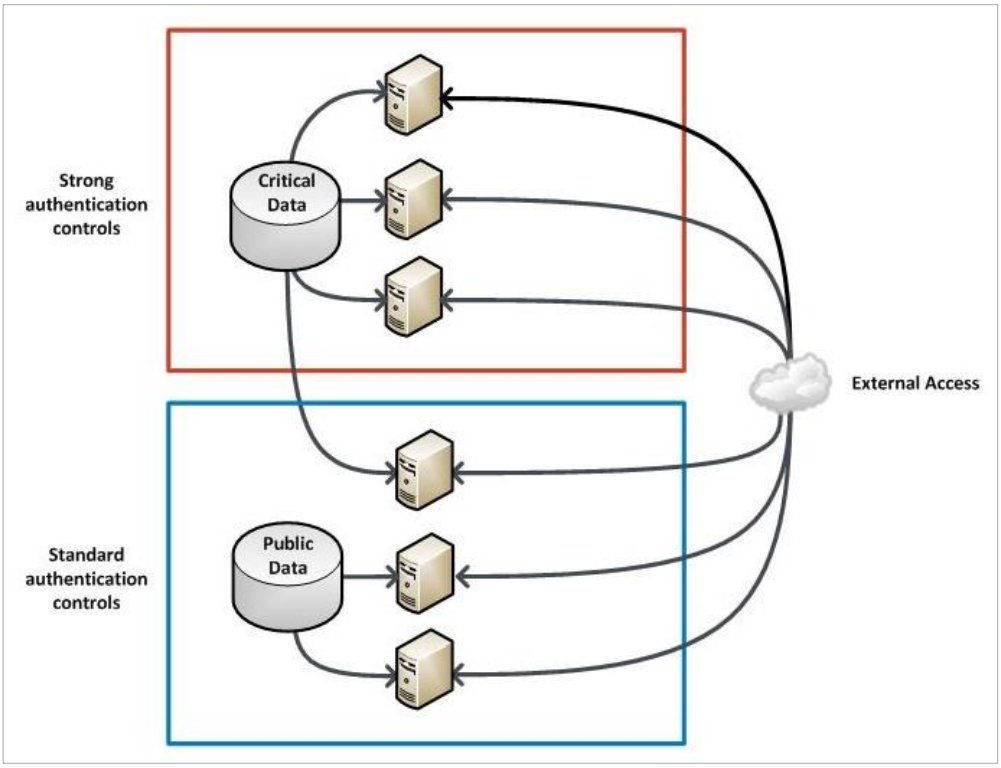

FIGURE 1 - Existing authentication boundaries

The lack of strong authentication controls allowed the assessment team to compromise critical data and more easily compromise systems without being detected by Acme security personnel.

Affected Locations

Total Instances **Systemic**

Recommendations

The assessment team recommends the following actions to mitigate the risks of inconsistent authentication boundaries:

- Re-evaluate existing network segmentation and supporting authentication controls on a per application basis, based on access to critical data. The recommended new architecture is shown below:

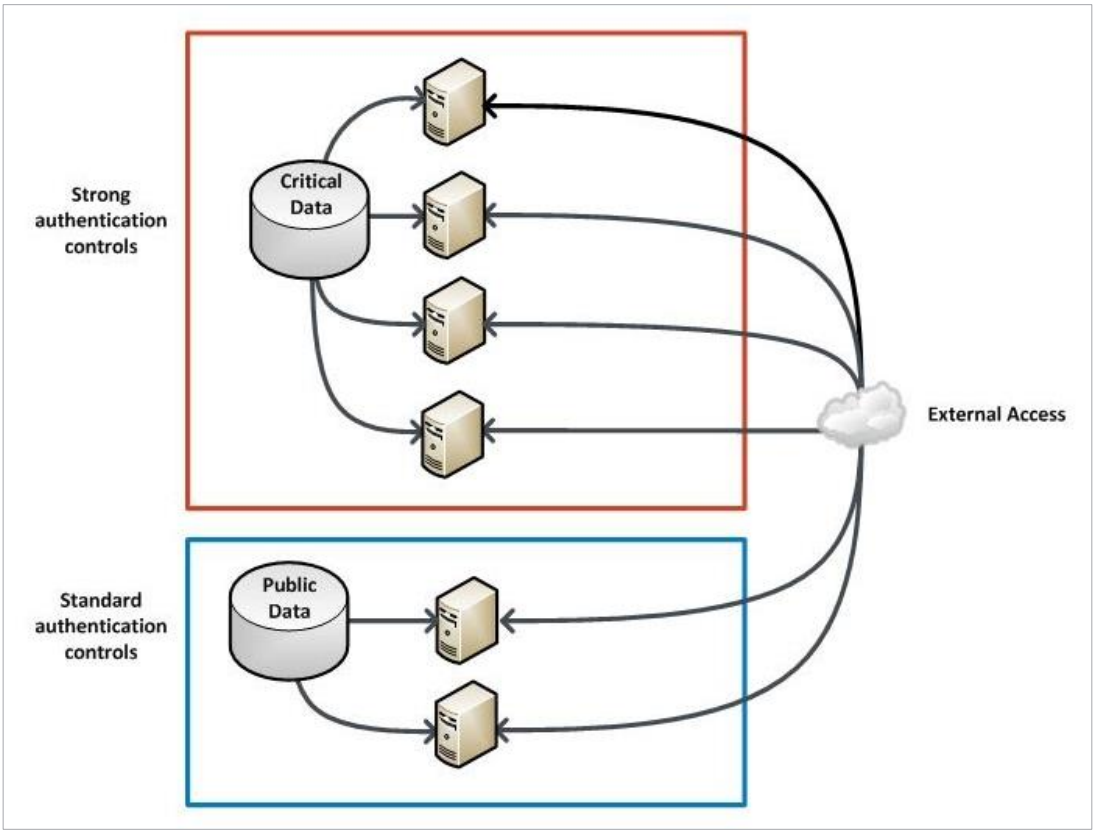

FIGURE 2 - Proposed authentication boundaries

- Add a checkpoint during the design phase of any new application or internal development to identify which access controls would be appropriate for any new applications based on the data those applications access. This prevents new applications from subverting existing security practices.

Additional Resources

The Perimeter Is Dead: Security Without Boundaries
https://www.securityroundtable.org/security-without-boundaries-perimeter-dead/

Product Security Review

The assessment team conducted a product security review with the following targets in scope:

- StoreOps POS application
- Host-based review of POS workstation

Identified Issues

2 INSECURE WORKSTATION DEPLOYMENT CRITICAL

Definition

Insecure workstation deployment occurs when there is a lack of security controls required to prevent unauthorized access to sensitive data, network resources, and functionality, which can allow attackers to obtain local administrator access to the system.

Details

The assessment team gained full control of the point-of-sale (POS) workstation in the retail location by leveraging the publicly available Windows password bypass tool Kon-Boot. Kon-Boot is a commercial application that bypasses the authentication process of Windows-based operating systems.

First, the assessment team interrupted the startup process and modified the BIOS boot order configuration. The team changed the boot order to load and run USB drives before the main hard drive. Then the team inserted a USB drive with an image of Kon-Boot and restarted the workstation. Kon-Boot's startup process running on the POS is shown below:

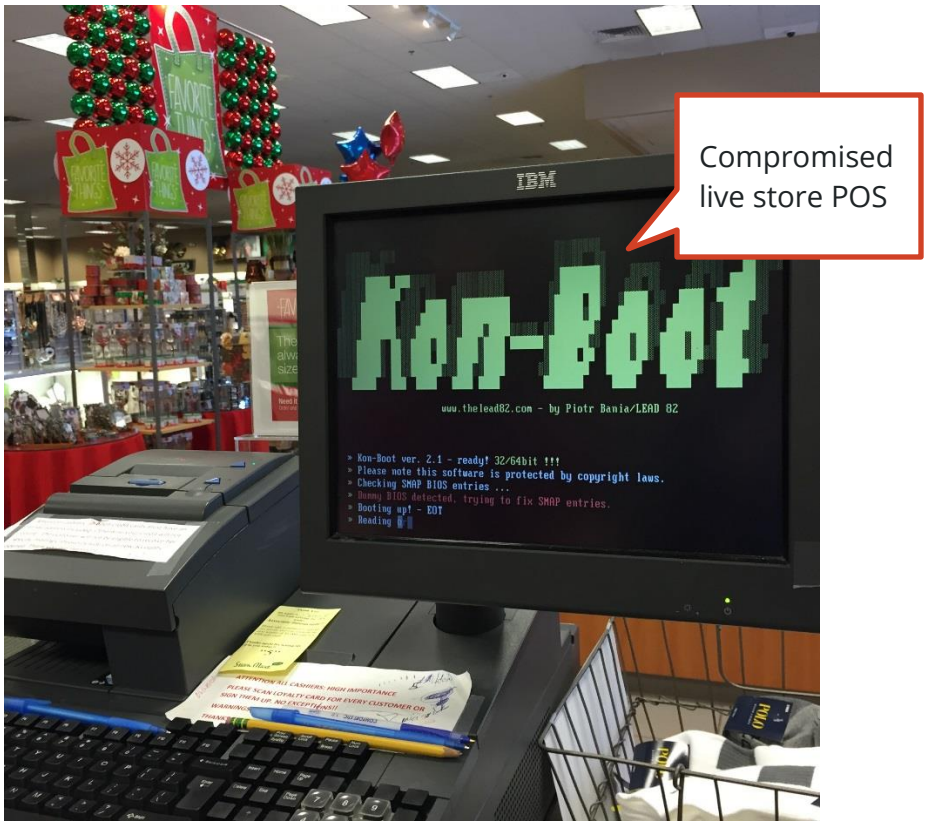

FIGURE 3 - Kon-Boot used via bootable USB

Use the following steps to reproduce gaining `NT AUTHORITY\SYSTEM` privileges on the target machine:

- Boot the Microsoft Windows operating system with Kon-Boot.
- Press **Shift** five times to launch (run) `cmd.exe`.
- Execute (run) the following command in the console from the writable directory default Windows directory path:

```
copy c:\windows\system32\cmd.exe cmk.exe
```

- Execute (run) `cmk.exe`.

These steps gave the assessment team a command prompt running with `NT AUTHORITY\SYSTEM` privileges and allowed the team to install or modify any software on the system. The following figure shows the terminal with the host name `R009201`, which also had the IP address `10.30.92.131`:

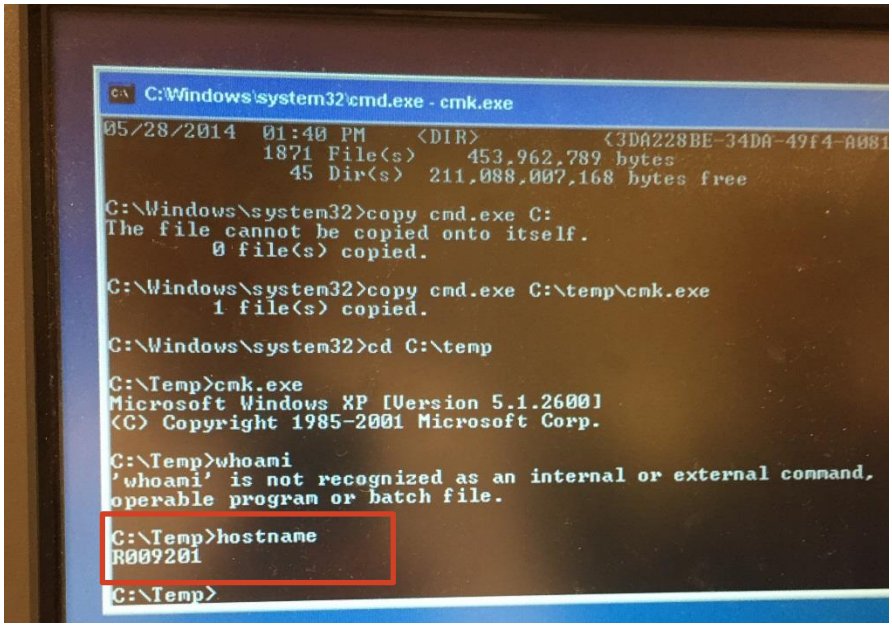

FIGURE 4 - Administrative level command prompt

The assessment team ran a meterpreter key logger payload on the POS workstation that captured valid cashier credentials to the POS application, as shown below:

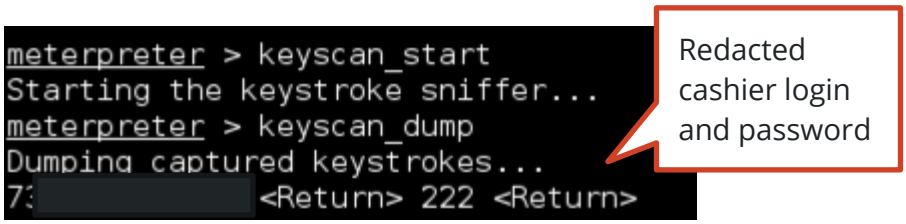

FIGURE 5 - Keylogger targeting a live POS

Additionally, the team extracted cleartext credentials from the system, including one belonging to a domain admin. The team used those credentials to gain access to the domain controller and extract 2,144 sets of domain credentials. For a complete list, please refer to the accompanying spreadsheet.

Affected Locations

Host Name
`R009201`

IP Address
`10.30.92.131`

Total Instances **2**

Recommendations

To remediate the insecure workstation deployment, the assessment team recommends the following actions:

- Use a full disk encryption protection solution.
- Disable booting from USB and CD in the BIOS of the machine and protect the BIOS with a password.

Additional Resources

Managing the Local Admin Password Headache
http://www.darkreading.com/risk/managing-the-local-admin-password-headache/d/d-id/1139373?

Kon-Boot
http://www.piotrbania.com/all/kon-boot/

Application Penetration Testing

The assessment team conducted an application penetration test with the following target in scope:

- `www.acme.com`

Identified Issues

3 WEAK CRYPTOGRAPHY HIGH

Definition

Weak cryptography occurs when an application improperly implements an accepted cryptographic algorithm, uses a custom cryptographic routine, insecurely calls validated cryptographic libraries, or makes calls to cryptographic libraries with known vulnerabilities.

Details

The assessment team discovered that the store settings determined whether a user's password was encrypted or hashed in the database. Unless there is a legitimate business case, it is insecure to store user passwords with a reversible encryption scheme. Regardless, the team found that best practices were not followed with either encrypted or hashed passwords.

The team identified insecure password storage on line 441 of the `StoreIdentity.cs` file, as shown below:

```
432.      // ============================================================
433.      // Determine how to handle the password based on the store
434.      // settings; checking whether scheme is Encryption or Hashing
435.      // ============================================================
436.
437.      var pwd = String.Empty;
438.
439.      // Encryption check
440.      if (CachedStore.Store.PasswordSecurityScheme.StartsWith("E"))
441.          pwd = Encryptor.Encrypt(criteria.Password.ToUpper(), 2);
442.
443.      // Hashing check
444.      if (CachedStore.Store.PasswordSecurityScheme.StartsWith("H"))
445.          pwd = Encryptor.SHA512Hash(criteria.Password);
```

FIGURE 6 - Password storage functionality from `DataPortal_Fetch()`

The `PasswordSecurityScheme` property defined whether a store used encryption or hashing for user password storage. In the case of encryption, the user password was

cast to uppercase before it was sent to the encryption function. Modifying all passwords to uppercase reduces the complexity and entropy, which makes it easier for an attacker to determine the cleartext password using frequency analysis or brute-force attacks.

Additionally, the `Acme.POS.Core.Security.Encryptor` class used a weak mono-alphabetic substitution cipher for storing user passwords. The key values used for the substitution cipher were hard-coded in the `Encryptor` class, as shown in the figure below:

```
446.      public class Encryptor
446.      {
447.          private Encryptor() { }
448.          private static readonly String[] Keys = new String[] {
449.      "ABCDEFGHIJKLMNOPQRSTUVWXYZ/0123456789abcdefghijlkmnopqrstuvwxyz!@#$%^&
   *()_-,.':;<>=|\\\"",
450.      "A5N8WX096EBKVOQJHRY1DIPTM42FZCL3GSU7
   !@z#y$x%w^v&u*t(s)r_q+ponmkljihgfedcba\\,.':;<>=|-\"_",
451.      "The secret luggage pass code is 12345",
452.      …omitted for brevity…
```

FIGURE 7 - Hard-coded encryption keys

The key highlighted above was used for all user password encryption per the hard-coded value `(2)` passed as the `keyIndex` value to the `Encryptor.Encrypt()` function. The complexity of the encryption was reduced further because the custom algorithm encrypted `/ ? [ ] ` { } ~ <space 0x20>` characters and any Unicode characters as `[?]`. Specifics of how the Acme system converts cleartext passwords using a simple substitution algorithm are documented line by line in the code comments below:

```
116.      /// <summary>
453.      /// Encrypt a string based on the given key index
454.      /// </summary>
455.      /// <param name="clearText">The string to encrypt</param>
456.      /// <param name="keyIndex">The index of the key used to encrypt (key =
   0 - no encryption)</param>
457.      /// <returns>The encrypted String</returns>
458.      public static String Encrypt(String clearText, int keyIndex)
459.      {
460.          if (clearText.Length == 0) return clearText;// 0 length strings are
   just returned
461.          StringBuilder encryptedText = new StringBuilder(clearText.Length);
   // we use a stringbuilder
462.          int _charIndex;
463.          char _char;
464.          foreach (char c in clearText)   // look at each unencrypted
   character
465.          {
466.      _charIndex = Keys[keyIndex].IndexOf(c); // get the chars index in the
   desired key string
```

```
467.        _char = (_charIndex == -1) ? '?' : Keys[0][_charIndex];    // return
    the corresponding char from the base key or '?' if out of range
468.        encryptedText.Append(_char);    // append the char to the output string
469.            }
470.            return encryptedText.ToString();    // return the encrypted string
471.        }
```

FIGURE 8 - Weak substitution cipher used to encrypt user passwords

It was also possible to decrypt a user password without any knowledge of the encryption scheme by using frequency analysis. This would be possible even if the passwords did not contain English words because the custom encryption routine reused the same key space and padding.

The assessment team gained access to the 98 encrypted passwords in the database and found that several users had the same password:

Username	Password	Cleartext
Daffy	9/HRKF	ABC123
Bugs	9/HRKF	ABC123
Tweety	9/HRKF	ABC123
Taz	9/HRKF	ABC123
Sylvester	9/HRKF	ABC123

FIGURE 9 - Decrypted passwords from the `Security.Login` database table

Additionally, an attacker with access to the application DLLs or application memory space could retrieve the hard-coded encryption keys and logic used in the substitution cipher.

When store settings dictated that user passwords be stored using a SHA512 hashing algorithm, the passwords were more secure than those that were encrypted, but they still did not follow best practices for password security.

Affected Locations

Lines of Code

- line 441 of the `StoreIdentity.cs file`
- lines 466-468 of the `Acme.POS.Core.Security.Encryptor`

Total Instances 2

Recommendations

To properly secure sensitive data, the assessment team recommends the following actions:

- Require that all use of cryptography involve peer-reviewed implementations of industry-accepted algorithms such as PBKDF2, which is recommended by NIST. Ensure that the implementation of these libraries within the application be performed in a secure fashion according to the specific algorithm's best practice guidelines.
    - First, combine the cleartext password with a unique salt value. This salt should contain 128 bits of cryptographically random data and be stored in the same table as the hashed password. The salt should be uniquely generated every time a password is stored and should not be shared between accounts or password instances.
    - Second, pass the combined value through a one-way hashing function and store the result as the protected value in the back-end data store.
    - Finally, when a user goes through the authentication process, ensure the application adds the unique salt value to the password and passes the combined value into the hash function. The result of the hash function should be compared to the hash stored in the back-end data store. If the hashes match, the user will be successfully authenticated.
- Ensure passwords are changeable no more than once per day to prevent users from intentionally defeating the password history system.
- When passwords are used on the client or the server, store them in non-garbage-collected character arrays, which can be explicitly overwritten by the system. As soon as a password is no longer needed, it should be overwritten.

Additional Resources

Mono-alphabetic Cipher Solver
http://www.secretcodebreaker.com/scbsolvr.html

MSDN - Crypto.HashPassword
http://msdn.microsoft.com/en-us/library/system.web.helpers.crypto.hashpassword(v=vs.99).aspx

MSDN - Rfc2898DeriveBytes Class
http://msdn.microsoft.com/en-us/library/system.security.cryptography.rfc2898derivebytes.aspx

Mobile Application Assessment

The assessment team performed a mobile application assessment with the following target in scope:

- `www.acme.com` iOS application

Identified Issues

4 CREDIT CARD PAN INTERCEPTION — HIGH

Definition

Payment processing systems are tasked with keeping cardholder data secure while in transit and at rest. Interception attacks target data in transit and, if successfully exploited, can result in the disclosure of sensitive cardholder information to an attacker.

Details

The assessment team identified a method that could be used to extract cardholder data and primary account numbers (PANs) from the Acme Anvil application without the knowledge of the payee.

During on-device penetration testing, the team found that non-swiped transactions left a PAN in memory long enough for it to be retrieved and exfiltrated on a specially configured iPad.

The team verified the exploit using a BF-extended version of a tool called Cycript, which enables anyone with a jailbroken iOS device to programmatically interact with the Objective-C runtime of a running iOS application. Cycript can be used to modify classes, add or change UI components, intercept and redirect Objective-C methods, read/write object properties and instance variables, and examine the application's heap and stack.

The team conducted the attack against a jailbroken iPad 2 running iOS 6.1. SSH was installed on the iPad and Cycript was run from the command line on the device. The highlighted text in the figure below represents commands entered by the team:

```
$ ./slcycript Acme\ Anvil
Importing JS functions...
Connecting to Cycript...
cy#
```

FIGURE 10 - Command used by team to attack jailbroken iPad

The `cy#` prompt is a JavaScript read-evaluate-print loop (REPL) into which the Objective-C runtime was merged, which made it possible to write JavaScript code that interacted with an iOS app's runtime in explorative and programmable ways.

While exploring the runtime, the team observed that non-swipe transaction data was handled by the `ManualEntryController` class, which was populated with either manually entered credit card digits or card data taken from the Card.io image recognition system. In either case, the masked data was displayed on screen until the **Next** button was pressed, as shown below:

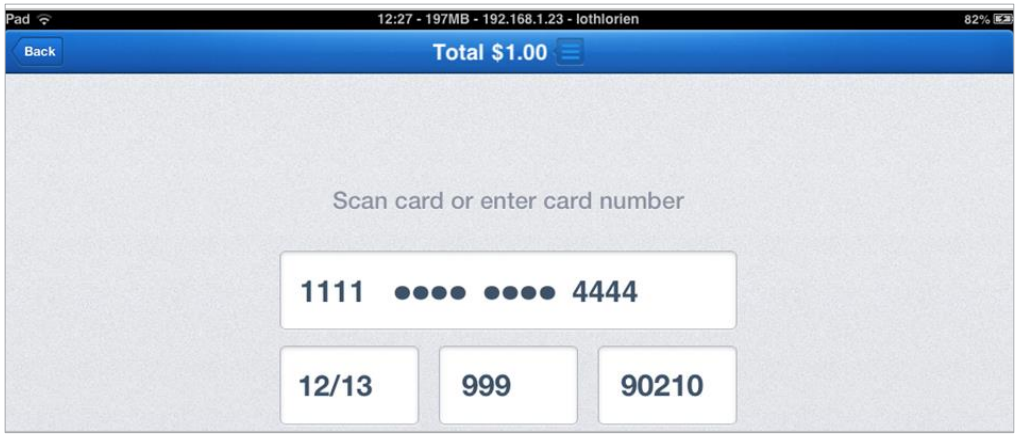

FIGURE 11 - Manually entered card data

To locate this data in the running app, the team used Cycript to parse `UIApplication.keyWindow.recursiveDescription` for the controls currently on screen and discovered an object called `PPHCardDataEntryView`:

```
<PPHCardDataEntryView: 0x1d96ca60; frame = (259 80; 500 300); layer = <CALayer:
0x1c545470>>
```

FIGURE 12 - `PPHCardDataEntryView` object

The team obtained a reference to the object and enumerated the card data, as shown in the figure below:

```
cy# pphEntry=new Instance(0x1d96ca60)
"<PPHCardDataEntryView: 0x1d96ca60; frame = (259 80; 500 300); layer = <CALayer:
0x1c545470>>"
cy# card=pphEntry->_cardData
"<TransientCardData: 0x1d9c8dc0>"
cy# card->cardNumber
"1111222233334444"
cy# card->cvv
"999"
cy# card->expirationMonth
12
```

```
cy# card->expirationYear
2013
```

FIGURE 13 - Reading credit card data from the `TransientCardData` object

`TransientCardData` is the class behind the card data on screen; the hex number `0x1d9c8dc0` is a pointer to the memory address of the current `TransientCardData` object. By attaching the object to the application using Cycript and then using Cycript to access internal classes, the team accessed the cleartext cardholder data in memory.

This attack would be straightforward to automate using `MobileSubstrate` tweaks to record credit card data entered manually during a transaction. The Card.io entry system was also affected due to the use a `TransientCardData` object to store Card.io card data.

Affected Locations

Application
iOS application

Total Instances **N/A**

Recommendations

To mitigate the risks of credit card PAN interception, the assessment team recommends the following actions:

- Avoid storing, processing, or transmitting unencrypted cardholder data on mobile devices at all times.
- Apply rigorous anti-jailbreaking countermeasures that are implemented in a low-level language such as C or assembly. The anti-jailbreaking routines should be obfuscated and updated with each release of Acme Anvil.
- Ensure that Acme's terms and conditions prohibit the use of jailbroken devices with Acme Anvil.
- Employ strict anti-debugging controls to make it harder to debug running instances of the application.
- Obfuscate all iOS classes, methods, properties, and instance variables in all release builds.
- Implement anti-tampering technology to prevent attackers from modifying the Objective-C runtime.

- Avoid storing sensitive data in class properties and instance variables, which are easily accessible to attackers. Instead, dynamically allocate and de-allocate storage for sensitive data at runtime. This raises the bar for attackers seeking to locate sensitive information in memory at runtime.

Additional Resources

Cycript
http://iphonedevwiki.net/index.php/Cycript

MobileSubstrate
http://iphonedevwiki.net/index.php/MobileSubstrate

Theos
http://iphonedevwiki.net/index.php/Theos

APPENDIX A — MEASUREMENT SCALES

The assessment team used the following criteria to rate the findings in this report. Bishop Fox derived these risk ratings from the industry and organizations such as OWASP.

Finding Severity

The severity of each finding in this report is independent. Finding severity ratings combine direct technical and business impact with the worst-case scenario in an attack chain. The more significant the impact and the fewer vulnerabilities that must be exploited to achieve that impact, the higher the severity.

Critical	Vulnerability is an otherwise high-severity issue with additional security implications that could lead to exceptional business impact. Examples include trivial exploit difficulty, business-critical data compromised, bypass of multiple security controls, direct violation of communicated security objectives, and large-scale vulnerability exposure.
High	Vulnerability may result in direct exposure including, but not limited to: the loss of application control, execution of malicious code, or compromise of underlying host systems. The issue may also create a breach in the confidentiality or integrity of sensitive business data, customer information, and administrative and user accounts. In some instances, this exposure may extend farther into the infrastructure beyond the data and systems associated with the application. Examples include parameter injection, denial of service, and cross-site scripting.
Medium	Vulnerability does not lead directly to the exposure of critical application functionality, sensitive business and customer data, or application credentials. However, it can be executed multiple times or leveraged in conjunction with another issue to cause direct exposure. Examples include brute-forcing and client-side input validation.
Low	Vulnerability may result in limited exposure of application control, sensitive business and customer data, or system information. This type of issue provides value only when combined with one or more issues of a higher risk classification. Examples include overly detailed error messages, the disclosure of system versioning information, and minor reliability issues.
Informational	Finding does not have a direct security impact but represents an opportunity to add an additional layer of security, is considered a best practice, or has the possibility of turning into an issue over time. Finding is a security-relevant observation that has no direct business impact or exploitability but may lead to exploitable vulnerabilities. Examples include poor communication between engineering organizations, documentation that encourages poor security practices, and lack of security training for developers.