SCADA And Mobile Security In The Internet Of Things Era

Alexander Bolshev (dark_k3y) Security Consultant, IOActive

Ivan Yushkevich (Steph) Information Security Auditor, Embedi

Abstract

Two years ago, the authors assessed 20 mobile applications that worked with ICS software and hardware. At that time, mobile technologies were widespread, but IoT mania was only beginning. In that paper, the authors stated, “convenience often wins over security. Nowadays, you can monitor (or even control!) your ICS from a brand-new Android [device].”

Today, the idea of putting logging, monitoring, and even supervisory/control functions in the cloud is not so farfetched. The purpose of this paper is to discuss how the landscape has evolved over the past two years and assess the security posture of SCADA systems and mobile applications in this new IoT era.

CONTENTS

Acronyms

Acronym	Definition
ACL	Access Control List
BYOD	Bring Your Own Device
DoS	Denial of Service
HMI	Human Machine Interface
ICS	Industrial Control System
IoT	Internet of Things
IIoT	Industrial IoT
MES	Manufacturing Execution System
MiTM	Man in The Middle
OPC	Open Platform Communications
PLC	Programmable Logic Controller A digital system used for automation of typically industrial electromechanical processes
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition Systems operating with coded signals over communication channels to provide control of remote equipment
SSRF	Server Side Request Forgery
XXE	XML eXternal Entity

Introduction

Two years ago, we assessed 20 mobile applications that worked with ICS software and hardware. At that time, mobile technologies were widespread, but Internet of Things (IoT) mania was only starting. Our research concluded the combination of SCADA systems and mobile applications had the potential to be a very dangerous and vulnerable cocktail. In the introduction of our paper, we stated "convenience often wins over security. Nowadays, you can monitor (or even control!) your ICS from a brand-new Android [device]."

Today, no one is surprised at the appearance of an IIoT. The idea of putting your logging, monitoring, and even supervisory/control functions in the cloud does not sound as crazy as it did several years ago. If you look at mobile application offerings today, many more ICS-related applications are available than two years ago. Previously, we predicted that the "rapidly growing mobile development environment" would redeem the past sins of SCADA systems. The purpose of this paper is to understand how the landscape has evolved and assess the security posture of SCADA systems and mobile applications in this new IIoT era.

SCADA and Mobile Applications

ICS infrastructures are heterogeneous by nature. They include several layers, each of which is dedicated to specific tasks. Figure 1 illustrates a typical ICS structure.

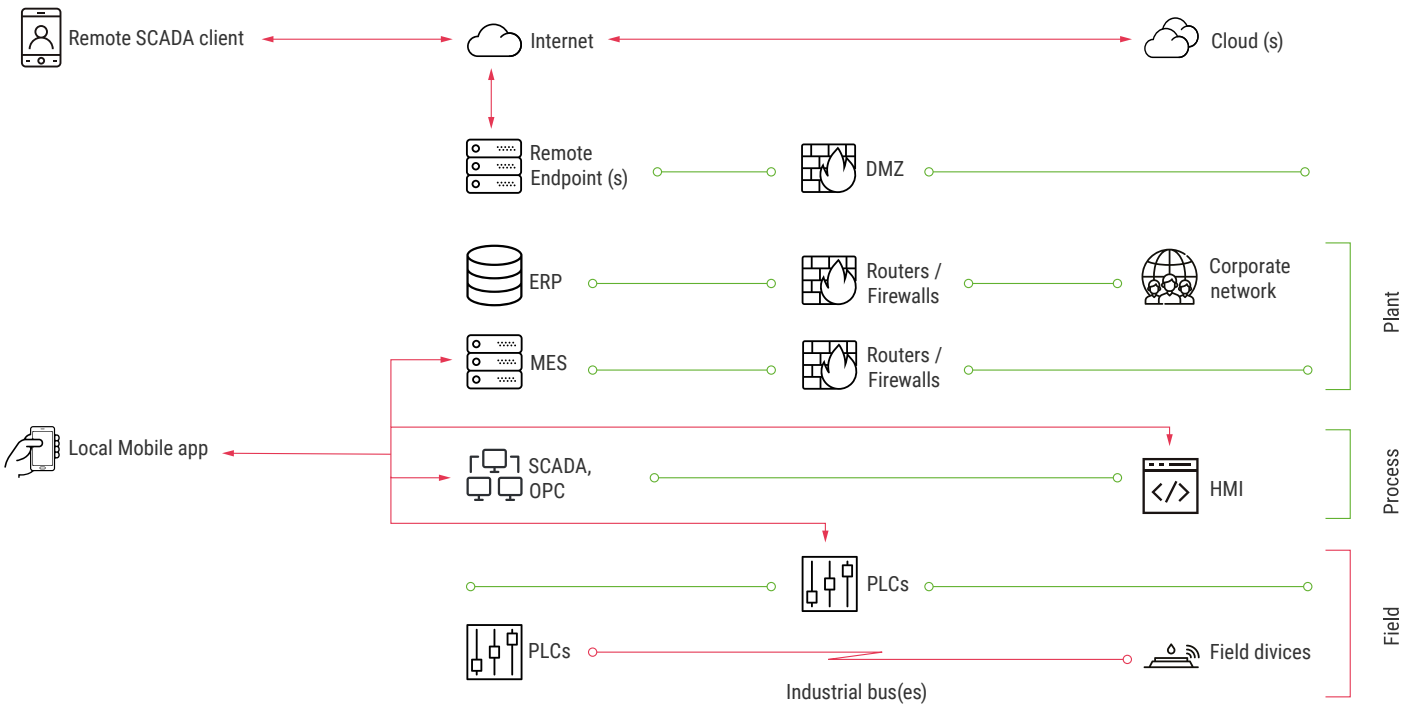

Figure 1: Modern ICS infrastructure including mobile apps

Mobile applications reside in several ICS segments and can be grouped into two general families: Local (control room) and Remote.

Local Applications

Local applications are installed on devices that connect directly to ICS devices in the field or process layers (over Wi-Fi, Bluetooth, or serial). For example, applications installed on tablets used by engineers working in a plant to view the current state of processes, even during their lunch break. Applications on such devices could be used as portable PLC/HMI configurators/programmers in place of a laptop. Such a solution would be more convenient to deploy due to the characteristics of the physical environment. We can divide local applications into the following subtypes:

- PLC configuration — An application for configuring or monitoring ICS devices, including PLCs, RTUs, industrial gateways, and switches.
- SCADA client — An application that connects to the SCADA server and allows engineers to view and/or supervise industrial processes.
- Mobile HMI panel — An application that transforms a mobile device into an actual HMI panel for the ICS device. This subtype is the most interesting (and potentially dangerous) control room application. Engineers could use pre-designed HMI projects uploaded using desktop applications or even design and program HMI projects from the devices themselves. After the project is uploaded and run on the mobile device, it turns the device into a real HMI panel (and can even fully replace wall-mounted devices that cost several thousands of dollars). The project connects to the process or field components using Modbus/TCP, OPC, S7 and other protocols, using Wi-Fi or Bluetooth as a physical medium.

One should note two environmental factors for every control-room application:

- The applications run on devices that reside inside "safe" (at least firewalled and separated control room networks. Thus, the lack of cryptography, authentication, and authorization of communication cannot be characterized as a medium or high risk. If devices inside the process network segment use Modbus/TCP, it is logical to assume that the mobile application leverages them as well.
- Even partial compromise of devices that reside in such a network segment (e.g., an attacker is able to execute custom code on the device) could allow attackers to directly/indirectly influence the industrial process. Thus, any interaction with such applications outside of the control room's borders poses a high risk. As a result, devices with such mobile applications rarely leave the control room or connect to external networks.

Remote Applications

Remote applications allow engineers to connect to ICS servers using remote channels, like the Internet, VPN-over-Internet, and private cell networks. Typically, they only allow monitoring of the industrial process; however, several applications allow the user to control/supervise the process. Applications of this type include remote SCADA clients, MES clients, and remote alert applications. In comparison to local applications belonging to the control room group, which usually operate in an isolated environment, remote applications are often installed on smartphones that use Internet connections or even on personal devices in organizations that have a BYOD policy. In other words, remote applications are more exposed and face different

attack types, like MiTM attacks over communication channels, or attacks from another malicious application that could be installed on the device.

Typical Threats And Attacks

In this section, we discuss the typical threats to this heterogeneous landscape of applications and how attacks could be conducted. We also map the threats to the application types.

Threat Types

There are three main possible threat types:

Unauthorized physical access to the device or "virtual" access to device data

This could occur in three scenarios:

U1. Device loss. In this case, attackers could gain access to the data in application storageby attacking the software and/or hardware. This could lead to data leakage. For local (control room) applications, leaking data could give attackers a more thorough understanding of the industrial process, ICS infrastructure, network addressing schemes, etc. For remote access applications, the consequences are much more dangerous: attackers could extract any authentication data stored on the device and use it to connect to remote SCADA endpoints.

U2. Unlocked device left unattended. Attackers could work in a timeframe of several minutes to extract or alter configuration/data in the mobile SCADA application.

U3. Unauthorized compromise of data on the partition without proper ACLs in place (for example, an SD card). If application data is stored on SD cards, attackers could gain access to or alter such data. Typically, SD cards are either external removable devices, internal storage partitions with the exFAT filesystem, or other filesystems without proper ACLs.

Communication channel compromise (MiTM)

For remote applications, the confidentiality and integrity of the communication channel to remote SCADA endpoints is crucial; however, devices often connect to the Internet using non-secure channels. This creates the following list of threats:

C1. Rogue Wi-Fi or GSM access points

C2. Public access point or network without proper security mechanisms

C3. Private (e.g., corporate or home) network compromise

C4. VPN channel compromise

Any of these threats could allow attackers to sniff, replay, or alter communication data between the application and remote SCADA endpoint.

Application compromise

Applications themselves could include various vulnerabilities on the:

A1. Server (backend) side

A2. Client side

This could lead to various vulnerabilities. For example, issues on backend services could include ACL issues/incorrect permission checking, remote code/command execution, insufficient data validation, or information leakage.

Table 1 summarizes the threat types.

Code	Threat Name
U: Unauthorized physical access to the device or "virtual" access to device data	
U2	Unlocked device left unattended
U3	Unauthorized compromise of data on the partition without proper ACLs in place (for example, an SD card)
C: Communication channel compromise	
C1	Rogue Wi-Fi or GSM access points
C2	Public AP or network without proper security mechanism
C3	Private (e.g., corporate or home) network compromise
C4	VPN channel compromise
A: Application compromise	
A1	Server (backend) side compromise
A2	Client side compromise

Table 1: SCADA mobile client threat list

Attack Types

Based on the threats listed above, attacks targeting mobile SCADA applications can be sorted into two groups.

Directly/indirectly influencing an industrial process or industrial network infrastructure

This type of attack could be carried out by sending data that would be carried over to the field segment devices. Various methods could be used to achieve this, including bypassing ACL/permissions checks, accessing credentials with the required privileges, or bypassing data validation.

Such attacks could exploit the following scenarios:

- Acting as a MiTM (C1-4) over an insecure communication channel, an attacker alters commands from a mobile SCADA application to the remote endpoint, which reaches the field devices.
- Attackers steal a device (U1) and extract remote SCADA endpoint credentials from it. Using them, they connect to the SCADA environment and send malicious commands. Alternatively, the attackers just take photos of the application's settings (including credentials) when the device is left unlocked and unattended (U2).
- Engineers unwillingly install a malicious application on their BYOD, which initially stays dormant to avoid raising any suspicion. Later, the malicious application exploits vulnerabilities in the victim application to subvert the communication process with the backend servers or extract valuable data (A2). Another possible case is when SCADA mobile applications store data on partitions with insufficient permission checking and the malicious application alters/reads it (U3).
- The backend servers are attacked using approaches from typical web or infrastructure application penetration testing (e.g., OWASP Top Ten risks) or by reverse-engineering the protocol between the mobile SCADA application and the remote endpoint. Then, the attackers leverage the vulnerabilities they have identified (A1) and send data to the backend servers, which will directly/indirectly influence some parts of industrial process/infrastructure.

Compromising a SCADA operator to unwillingly perform a harmful action on the system

The core idea is for the attacker to create environmental circumstances where a SCADA system operator could make incorrect decisions and trigger alarms or otherwise bring the system into a halt state. In other words, if attackers subvert the signal channels between the application and the actual system, they could confuse the SCADA operator about the current state of the system. This could be achieved by subverting the data going from the system to the HMI panel or mixing the signals going from the panel to field devices. Based on this maliciously altered information, SCADA operators could take harmful actions in good faith.

The following are example attack scenarios:

- Using MiTM (C1-4) over an insecure communication channel, attackers alter the data going from a SCADA system to a mobile application. This means that operators will see a completely different system state on their mobile screen.
- Attackers use physical access (U1), malicious applications (U3), or vulnerabilities in other applications installed onto the same device (A2) to alter the mobile HMI project database stored on the SD card, exploiting a lack of proper ACLs. This will allow attackers to completely subvert an operator's understanding of the system. For example, they could change displayed data, hide events, or subvert switch controls behavior (e.g., make the "OFF" switch execute an "ON" state and vice versa). Besides physical compromise and mobile virus, this also could be achieved using a ZIP traversal vulnerability (described below).

Testing Approach

Similar to the research we conducted two years ago, our analysis and testing approach was based on the OWASP Mobile Top 10 2016. Each application was tested using the following steps:

- Perform analysis and fill out the test checklist
- Perform client and backend fuzzing
- If needed, perform deep analysis with reverse engineering

We did not alter the fuzzing approach since the last iteration of this research. It was discussed in depth in our previous whitepaper, so its description is omitted for brevity.

We improved our test checklist for this assessment. It includes:

- Application purpose, type, category, and basic information — This field consists of generic application characteristics, such as name, package name (namespace), vendor details, Google Play link, purpose, type, and subtype (see section 2).
- Permissions — Required by applications. Determined by analyzing the Android Manifest for instance.
- Password protection — In the case of unlocked and unattended devices, additional password protections are very useful and could prevent attackers from gaining any useful information from the device. Also, the password(s) could be used as a key component for encrypting application data.
- Application intents, exported providers, broadcast services, etc. — Any external data command channel that the application exposes increases its attack surface and could be leveraged by a malicious application. For instance, incorrectly exported content providers could lead to data leaks or tampering; or SQL injection in broadcast intent could be used by an attacker to compromise the application's behavior.

- Native code — The presence of native and unmanaged code could present a large attack surface and allow attackers to exploit low-level vulnerabilities, such as memory corruptions and buffer overflows, which are typically mitigated by Java-managed code by design.
- Code obfuscation — Obfuscated code can make reverse engineering and security assessments of applications much more difficult and time consuming. While it would not stop attackers, it could greatly slow them down.
- Presence of web-based components — Web-based components in the application's interface could expose a very dangerous attack surface, especially in the case where the attackers find a way to inject custom JS/CSS or HTML code. This can be even worse, as a WebView with JavaScript enabled can lead to remote code execution on versions of Android before 4.2.
- Methods of authentication used to communicate with the backend — Correct authentication between the application and backend is extremely important. A flawed authentication method could allow attackers to gain unauthorized access to the backend. For instance, if authentication is only enforced client-side, then attackers can bypass the login form on the application by communicating directly with the backend.
- Correctness of operations with sessions, cookies, and tokens — Improper session handling or insecure session generation could allow attackers to impersonate a legitimate operator, for example, by performing session fixation attacks or brute-forcing the operator session ID.
- SSL/TLS connection configuration — Incorrect settings when establishing TLS connections could lead to various MiTM attacks, via downgrade attacks, for instance.
- XML parser configuration — If the application handles XML data, a vulnerable parser or incorrect XML parser settings could lead to various XXE and SSRF vulnerabilities.
- Backend APIs — Incorrectly implemented backend APIs may expose internal or debug functionalities to an attacker. Errors in input validations may also be exploited, for example, to remotely execute code on the backend servers.
- Sensitive data handling — Storing sensitive data, such as credentials, tokens, or keys, on unsafe storage could provide a significant advantage to attackers.
- HMI project data handling — If the HMI project stores its data with insufficient permissions, an operator could be misled into taking harmful actions by compromising the HMI project.
- Secure storage — Storing data at rest without enforcing proper cryptographic secrecy and integrity could allow an attacker to tamper with or read such data.
- Other issues — Any other issue that was discovered during the analysis presented in this paper.

In the following table, we correlate the OWASP Top Ten Mobile 2016 issues with possible attack vectors against SCADA Mobile applications

OWASP ID	Description	Threats
M1	Improper Platform Usage	A2
M2	Insecure Data Storage	U1, U3, A2
M3	Insecure Communication	C1-4
M4	Insecure Authentication	C1-4
M5	Insufficient Cryptography	U1-3, C1-4
M6	Insecure Authorization	A1, U2
M7	Client Code Quality	A1-2, U1-2
M8	Code Tampering	U1
M9	Reverse Engineering	-
M10	Extraneous Functionality	A2

Table 2. Mapping OWASP IDs to threats

Reviewed Vendors

Table 3 lists the 34 vendors we analyzed in our research. It includes the vendor name as well as the server software and type. We randomly selected SCADA application samples from the Google Play Store. We did, however, favor applications for which we were granted access to the backend hardware or software, so that a wider attack surface could be tested. Additionally, we excluded applications whose most recent update was before June 2015, since they were likely the subject of our previous work. We only retested them if there had been an update during the subsequent two years.

Vendor	Server/Backend/Protocol	Type
BACmove	BACnet	Local
BACmove	BACnet	Local
Cybrotech	CyBro Controllers, Modbus	Remote
Riccardo Pretolesi	Modbus/TCP	Local
Alexander Maier	KNX/EIB Devices	Remote
ZRA Ellat, SIA	Ellat SCADA	Remote
Fernhill Software	Fernhill SCADA Server	Local
IDEA-Teknik	Modbus, S7, …	Local
HENNEUSE ALAIN	Siemens Logo, 1200, …	Local
HENNEUSE ALAIN	Modbus	Local
Schneider Electric SE	IGSS SCADA Plants	Remote
Andrew L.	Modbus	Local
ISW Industriesoftware GmbH	S7	Local
ICONICS	Web Backend	Remote
Mita-Teknik	Web Backend	Remote
Planet Sol3a Systems	Modbus	Local
Progea	Movicon SCADA Server	Local
Progea	Movicon SCADA Server	Remote
mySCADA Technologies	Modbus, S7, …	Local
Kassl GmbH	OPC DA	Remote
ASEM S.p.A.	ASEM SCADA servers	Local
XCV Software	Modbus, Omron, S7, …	Remote
JC Accounting & Innovative Technologies, Inc	Modbus, A-B, S7, …	Local

Vendor	Server/Backend/Protocol	Type
Digital Electronics Corporation	Various HMI panels from Pro-Face	Local
Prosys OPC	Prosys OPC US Server	Local
Prumsys	Siemens S7 Family	Local
Appliworld	Modbus/TCP	Local
Siemens AG	Siemens LOGO Plc	Local
Siemens AG	Siemens S7 Family	Local
Beijer Electronics Products AB	SmartD HMI Device	Remote
TeslaSCADA	Modbus, S7, ControlLogix, OPC …	Local
TeslaSCADA	Modbus, S7, ControlLogix, OPC …	Local
Neodian Technology Co., LTD	Web Backend	Local
WHS	Web Backend	Remote

Table 3. Reviewed vendors

Analysis Of Finding

We identified 147 security issues in the applications and their backends. We classified each issue according to the OWASP Top Ten Mobile risks and added one additional category for backend software bugs.

Table 4 presents the distribution of findings across categories. The “Number of Issues” column reports the number of issues belonging to each category, while the “% of Apps” column reports how many applications have at least one vulnerability belonging to each category.

OWASP ID	OWASP Category/Type	Number of Issues	% of Apps
M1	Improper Platform Usage	5	6%
M2	Insecure Data Storage	20	47%
M3	Insecure Communication	11	38%
M4	Insecure Authentication	6	18%
M5	Insufficient Cryptography	8	24%
M6	Insecure Authorization	20	59%
M7	Client Code Quality	12	35%
M8	Code Tampering	32	94%
M9	Reverse Engineering	18	53%
M10	Extraneous Functionality	8	24%
	Backend Issues	7	12%

Table 4. Vulnerabilities statistics

The following sections provide an overview of each category, along with examples of the most significant vulnerabilities we identified.

M1: Improper Platform Usage

This category covers misuse of a platform feature or failure to use platform security controls.

During our research we uncovered several cases of improper platform usage. Four were related to outdated runtime libraries, and one was related to incorrect intent exporting.

Several applications used an old Xamarin Engine that is subject to a DLL hijack vulnerability. An attacker with write access to the SD Card could substitute Monodroid engine's DLLs, as well as the application's DLLs, with malicious ones. This would lead to malicious code execution in the context of the application. This issue was quickly identified when reviewing the device logs during application startup:

```
02-22 09:30:22.851 17010 17010 W monodroid: Using override path:
/data/user/0/com._____________/files/.__override__
02-22 09:30:22.851 17010 17010 W monodroid: Using override path:
/storage/emulated/0/Android/data/com._____________/files/.__override_
```

An attacker could inject custom malicious code into one of the application DLL files and upload it to the SD card folder "/storage/emulated/0/Android/data/com._____ /files/.__ override__". This could be achieved, for example, by another application, using an external device, or exploiting a ZIP traversal vulnerability in another app (U2-U3 or A2). A successful exploitation of the issue is shown below:

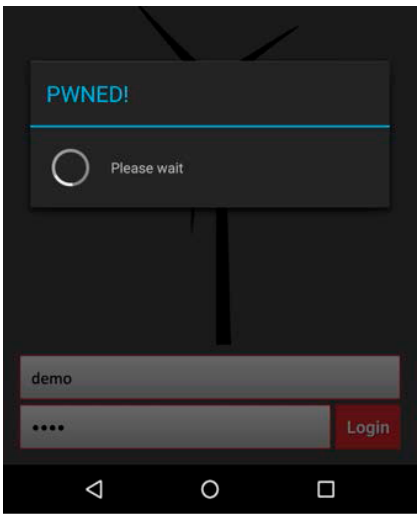

Figure 2: Xamarin DLL Hijack Exploitation

This attack fully compromises the application and could be used to pivot the next stage of the attack against the backend servers or mislead the SCADA operator.

We also identified incorrect exports of intents (data providers) in other applications. In this case, the vulnerable application sets up a content provider, which is only meant to be used internally. However, due to an incorrect Android manifest configuration, this provider is exported without any restrictions. This allows any application running on the same device to read and modify data in all the databases (which includes project data, communications data, settings, passwords, etc.).

This issue was discovered using the drozer tool:

```
dz> run app.provider.info -a _____________________.scada
Package: _____________________.scada
Authority:           .service.provider.
Read Permission: null
Write Permission: null
Content Provider:          .    .provider.
Multiprocess Allowed: False
Grant Uri Permissions: False
```

This provider allows full access to the following schemas of the internal database:

```
dz> run app.provider.finduri _____________________.scada
Scanning                                  .scada...
content://__________.___.provider._____________________/objects
...
content://__________.___.provider._____________________/servers/
content://__________.___.provider._____________________/screens/
...
content://__________.___.provider._____________________/settings
content://__________.___.provider._____________________/events
content://__________.___.provider._____________________/settings/
content://__________.___.provider._____________________/tags
```

By default, from Android 4.2 onwards, content-providers are not automatically exported. However, this application explicitly sets <uses-sdk minSdkVersion="9"> in its manifest, which causes export-by-default of all application content providers.

Any other application installed on the same device could read the admin password:

```
$ content query --uri content://__________.___.provider.
_____________________/settings/
Row: 0 startscreen=0, res2=, certificate=*********,
username=User, timeinterval=1000, adminpass=123, res1=1,
userpass=, debug=1, _id=1, orientation=512, adminname=Admin
```

Also, any other application can modify the server URL to redirect to a malicious server:

```
$ content query --uri
content://          .    .provider.        /servers
Row: 0 password=1000, res2=, display=Server, username=Unknown,
mode=5502, url=192.168.1.34, policy=0, res1=0, _id=1, anonymous=1
$ content update --uri content://
__________.___.provider._______/servers --bind url:s:192.168.1.100
--where "display='Server'"
$ content query --uri
content://          .    .provider.        /servers
Row: 0 password=1000, res2=, display=Server, username=Unknown,
mode=5502, url=192.168.1.100, policy=0, res1=0, _id=1,
anonymous=1
```

In general, other applications can access and modify all the data exported by the content providers (settings, project data, passwords, communication data, events). Similar to the previous scenario, an attacker can fully compromise the application, target the operator, and pivot to the backend servers using the credentials.

M2: Insecure Data Storage

This covers insecure data storage and unintended data leakage

Around half of the applications we analyzed (47%) were susceptible to Insecure Data Storage vulnerabilities. All of the affected applications were storing data on an SD card (external) or on the virtual (emulated) storage partition. As a side effect, these applications inherited the weaknesses of the filesystems used by these storage devices, as they have no proper ACLs or permission mechanisms implemented. In other words, if the application has the privileges to read/write to this device, it has full access to other data stored on the same device by other applications. Also, SD cards can be removed by a local attacker who can copy the data without the user's knowledge if the device is left unattended. The attacker would gain access to valuable data from the device (credentials, endpoints, HMI projects, logs, etc.) and could use it to mount further attacks. In addition, the attacker could also use that opportunity to tamper with the data (e.g., connection strings, HMI projects) to taint the operator's view of the system or enable further attacks using other vulnerabilities (see M1 and M7 risks).

The following screenshot shows the official application documentation, which requires the configuration file to be kept in the device's local storage.

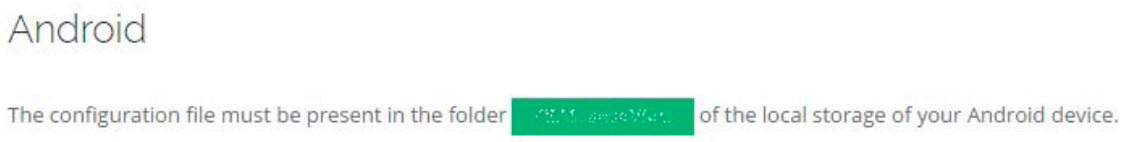

Figure 3: Documentation of Mandatory Location of Sensitive Files

M3: Insecure Communication

This covers poor handshaking, incorrect SSL versions, weak negotiation, cleartext communication of sensitive assets, etc.

Over one-third of all analyzed applications (38%) do not properly configure or use secure communications. We did not include applications using Modbus and other ICS-related protocols in this category, which by design lack communication security: the mobile application cannot use secure channels if the ICS device does not support encryption. Instead, we focused on applications that attempt to use secure communication, but fail to do so properly or fail to prevent unencrypted communication when they are outside of the isolated ICS perimeter. To be exhaustive, we reviewed direct communications with backend servers as well as any other communication channel used by the application, such as update mechanisms, telemetry, etc.

The most common issue we identified is related to the lack of TLS certificate validation. We found that no checks were performed by the applications to ensure that they were communicating with the genuine backend and not a rogue backend server.

Only 10% of the applications that use encrypted connections securely implement certificate validation (see Figure 4).

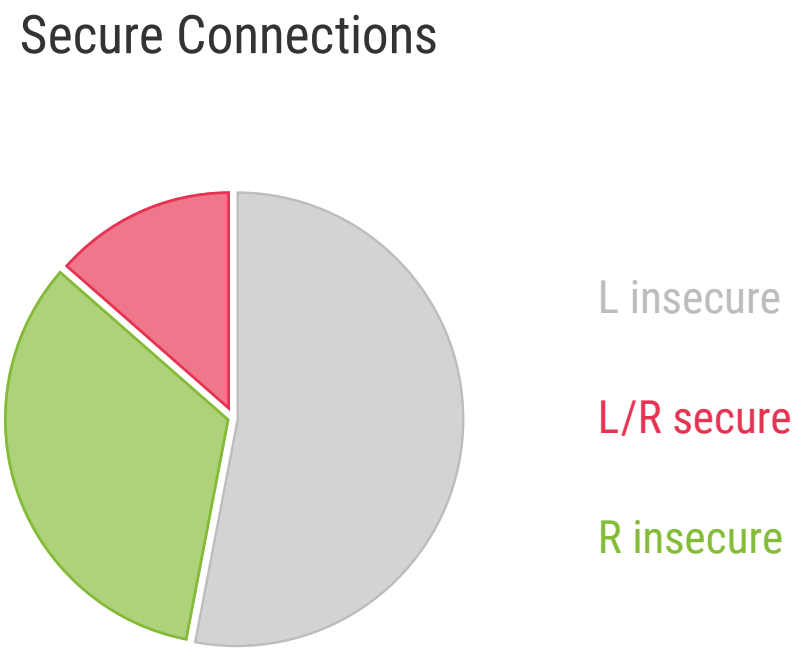

Figure 4: Security Posture of Communication for L(ocal) and R(emote) SCADA Mobile Applications

Nowadays, mechanisms for certificate validation are part of every modern operating system. However, applications sometimes implement their own mechanisms (at least five of the reviewed applications attempted to do so), which can be very complex. Also, certificate data for pinning is stored in the application's folder (and could be updated dynamically), creating additional problems for developers. Such an approach could lead to unintentional mistakes, therefore, using well-known existing mechanisms is recommended.

Not every application that implements a custom certificate validation does so correctly. For example, one of the reviewed applications just checked that the certificate's issuer name was the same as the application vendor's name.

```
If(array[0].getIssuerX500Principal().getName().contains("*****SCADA"))
return true;
```

In this case, an attacker could generate his own certificate using the vendor's name in the 'Issue' field and use it to conduct MiTM attacks.

Additional vulnerabilities were related to failing to prevent insecure communication when outside of the isolated ICS environment. For example, three applications always used cleartext HTTP to communicate and authenticate with their backend.

```
1) login?UserName=admin&Password=admin&IsMobile=true
2) ServerSummary?user=demo&pw=demo
3) POST /service/check_username_password/ HTTP/1.1
Content-Length: 29
username=guest&password=guest
```

If such applications are used outside of their safe perimeter, it will be easy for an attacker to acquire credentials and use them to establish a connection to the remote endpoints.

We also found cases where attackers could downgrade from secure to insecure connections or use non-suitable protocols for secure communications, as shown in the examples below.

One of the applications would fallback from HTTPS to cleartext HTTP when it failed to establish a secure communication channel (e.g., connection refused or invalid server certificate):

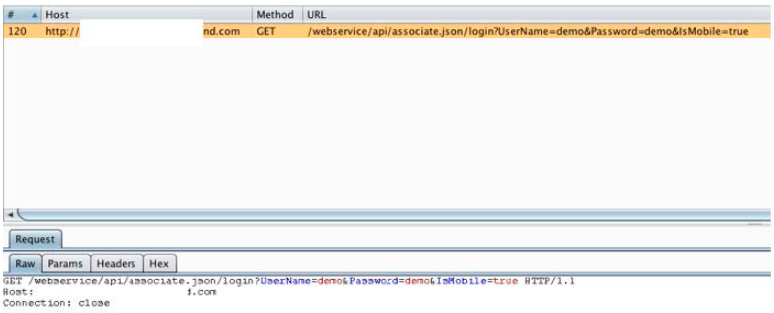

Figure 5: HTTPS to HTTP Connection Downgrade

Inevitably, such a fallback mechanism can be exploited by an attacker, who could simply block TLS connections, wait for the application to fallback to cleartext communication, and place themselves in a MiTM position to eavesdrop and even tamper with the data being transmitted.

Another application used a communication protocol that may not be best suited for the tasks it was performing. The surveyed application implements an HMI panel, and receives HMI project data from the desktop development application. Any time a mobile application project needs to be updated, the mobile application starts an FTP server, and allows the desktop IDE to upload and download files via an FTP client, as can be seen in Figure 6.

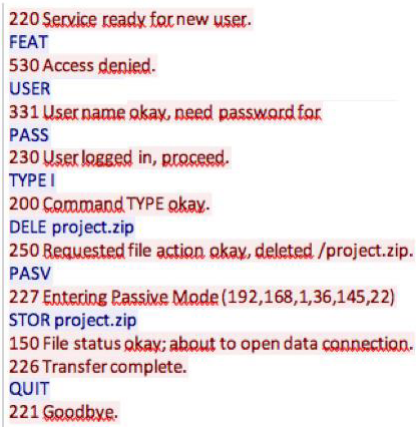

Figure 6: HMI Project Upload Using FTP

The main issue here, apart from the lack of an encryption layer, is that the authentication credentials for the FTP server were hardcoded in the mobile application's code and could not be changed. In other words, anybody on the same network as the mobile device and who reversed the application's code and extracted the credentials, could download/upload any data from the application, which could lead to successful execution of both the attacks described in section 3.

M4: Insecure Authentication

This category captures notions of authenticating the end user or bad session management.

About a fifth of the analyzed applications (18%) did not implement proper authentication. The main issues revolved around:

- Weaknesses in session management (e.g., storing device-specific authentication tokens on insecure storage)
- Absence of password checking when interacting with the device or server
- Storing passwords on the device using weak algorithms (e.g., using MD5 or even just base64 encoding)
- Failing to assess the user's identity for sensitive operation, including allowing anonymous sensitive requests to the backend when an authentication token would be expected

Another possible fault of insecure authentication lies in multi-user applications. One may wonder why mobile applications installed on a single-user operating system should implement multi-user functionality. In fact, HMI panel tablets are sometimes passed from one engineer to another inside the safe perimeter. Also, tablets can be used as a substitute for a traditional HMI panel. In this case, some kind of control on users and privileges could be required (e.g., a Senior Engineer should have more permissions than a Junior Engineer). We discovered a couple of applications that implement this approach incorrectly (e.g., allowing unauthorized users to access HMI schemes they do not own by simply specifying the scheme ID).

M5: Insufficient Cryptography

The code applies cryptography to a sensitive information asset. However, the cryptography is insufficient in some way.

Almost a quarter of the analyzed applications (24%) used cryptographic primitives incorrectly or in a weak manner. These applications used incorrect cryptographic approaches, weak cryptographic schemes, or hardcoded encryption keys.

We identified two applications that not only used a fixed Initialization Vector (IV) for symmetric encryption, but surprisingly, happened to use the same, pseudo-random hardcoded IV value. The code handling the IV was the same in both applications, as shown in Figure 7.

```
}

protected IvParameterSpec GetIv()
{
    byte[] array = new byte[this._writer.BlockSize];
    Array.Copy(Encoding.UTF8.GetBytes("fldsjfodasjifudslfjdsaofshaufihadsf"), 0, array, 0, this._writer.BlockSize);
    return new IvParameterSpec(array);
}

protected SecretKeySpec GetSecretKey(string key)
{
    byte[] key2 = null;
    try
    {
        key2 = this.CreateKeyBytes(key);
    }
    catch (UnsupportedEncodingException)
    {
    }
    catch (NoSuchAlgorithmException)
    {
    }
    return new SecretKeySpec(key2, "AES/CBC/PKCS5Padding");
```

Figure 7: IV Handling Code

By searching for this IV using the Google search engine, we were able to find it mentioned in many code snippets hosted on GitHub, as well as in StackOverflow.com answers (see Figure 8). The code snippet found on StackOverflow.com was almost identical to the code we extracted from the applications, which suggests that the developers used it as a reference. This violates the cryptographic requirement to choose a unique and unpredictable IV value for each encryption operation under the same key.

Figure 8: StackOverflow Answer Suggesting the Fixed IV

M6: Insecure Authorization

This is a category to capture any failures in authorization (e.g., authorization decisions in the client side, forced browsing, etc.).

We decided to include the lack of password protection in this category, rather than in "Insecure Authentication." In our case, the application password, restricting access to the HMI or the configuration, is more of a secondary verification of the user's abilities to use the application, rather than a system to assess an identity. In the initial state, the application deals with a partially-trusted user, and the application password is used to verify that the user could indeed be fully trusted. This also applies to passwords protecting the application's configuration (e.g., users could view system state, but need a password to supervise it).

Absence of password protection may allow attackers with access to unattended, unlocked devices to read/modify application configuration, leading to several issues, from extending the attacker's view on the system to malicious HMI project modification, or even leaking remote SCADA endpoint credentials. For more information on the topic, please refer to section 3.

Password protection can be used to achieve two goals:

1. Protecting the application's settings and data at runtime from being exfiltrated or modified
2. Protecting the application's sensitive data at rest using the password to securely derive an encryption key

As was mentioned before, if both goals are achieved, it could enable most of the threats from the U1-3 group.

Unfortunately, less than 20% of local applications implement secure authorization correctly. The most common mistake was the complete lack of passwords to protect the HMI project and panel data configuration – if a password was requested, it would only be used to protect the global application configuration. The second most common mistake was the presence of a "Do not ask for password again" feature, which defeats the purpose of runtime password protection.

M7: Client Code (Lack of) Quality

This would be the catch-all for code-level implementation problems in the mobile client.

During our analysis, we found multiple problems in client implementations: from DoS due to incorrect exception handling, to memory corruption vulnerabilities. Many applications implemented poor exception handling, which resulted in multiple crashes; this affected not only applications using native code, but also Java- and .Net-based ones. Native code applications also had various problems related to memory handling, where they failed to sanitize and prevent arbitrary read access violations (see Figure 9), and were additionally affected by memory corruption issues.

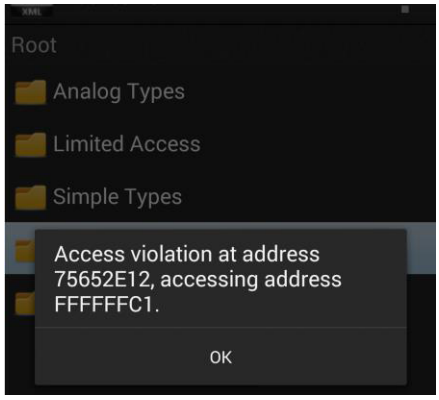

Figure 9: Access Violation in Application Using Native Code

Focusing on incorrect bounds-checking issues, an interesting case was an application that exposed a custom native-code HTTP server to all devices residing in the same network segment. This constitutes a very interesting attack surface for an attacker and a good example of M3/M6. As it turns out, the custom HTTP server was riddled with security issues, such as memory corruptions. For instance, simply supplying a very large (or negative) number in the Content-Length HTTP header caused the server to crash, as shown in Figure 10:

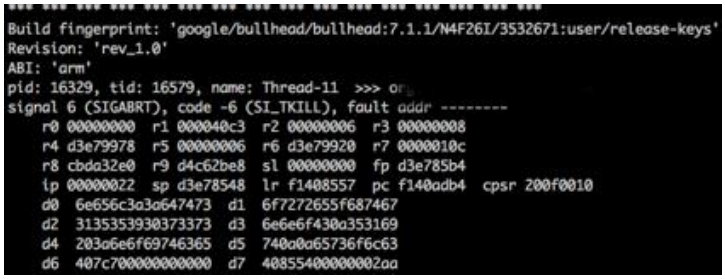

Figure 10: HMI Panel Application Crash due to Content-Length HTTP Header

Many HMI applications allow the user to upload, download, or save a project as a ZIP file, with further exporting to a server or another device. However, if ZIP archives are not handled correctly, applications could become vulnerable to several attacks, such as ZIP bombing or, more dangerously, directory traversal. A typical ZIP file structure is depicted in Figure 11.

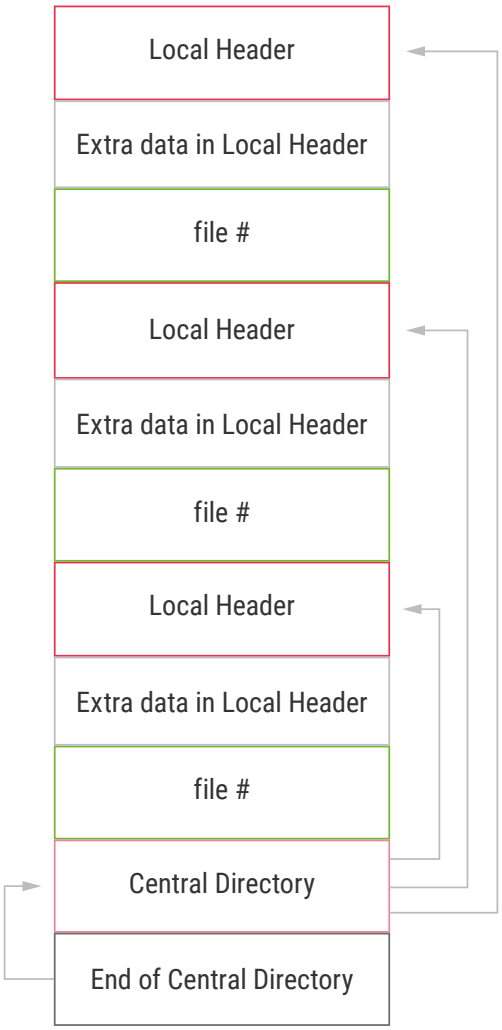

Figure 11: Typical ZIP File

Each Local Header field contains information about a compressed file (e.g., size, date of creation, and crucially, the filename). When unzipping occurs, the filename provided in the Local Header is combined with the path where the archive will be unzipped. If the process is not implemented correctly, it creates a dangerous opportunity for possible directory traversal. For example, if name in Local Header field is "../data.png", then the final path could be /data/data/app.name/database/../data.png, which resolves to /data/data/app.name/data.png. Thus, if an attacker could subvert the archive, he could rewrite any arbitrary file that is write accessible for the application.

A proof-of-concept attack was conducted using evilarc. We chose a mobile HMI panel that would download a zipped project from a remote web server without enforcing server authentication. Hence, the attacker can MiTM the client-server connection, and the mobile application may download a malicious ZIP archive designed to exploit path traversal. A malicious archive could be created as follows:


```
steph@              /evilarc $ python evilarc.py --os=unix --path=default -d 1 plan\ back.jpg
Creating evil.zip containing ../default/plan back.jpg
steph@              /evilarc $ sudo cp evil.zip /var/www/html/data/
```

Figure 12: Using evilarc to Create a Malicious ZIP File

After downloading and unpacking the project, the user would not notice any malfunction; however, the background image of another HMI project has been changed by the attacker.

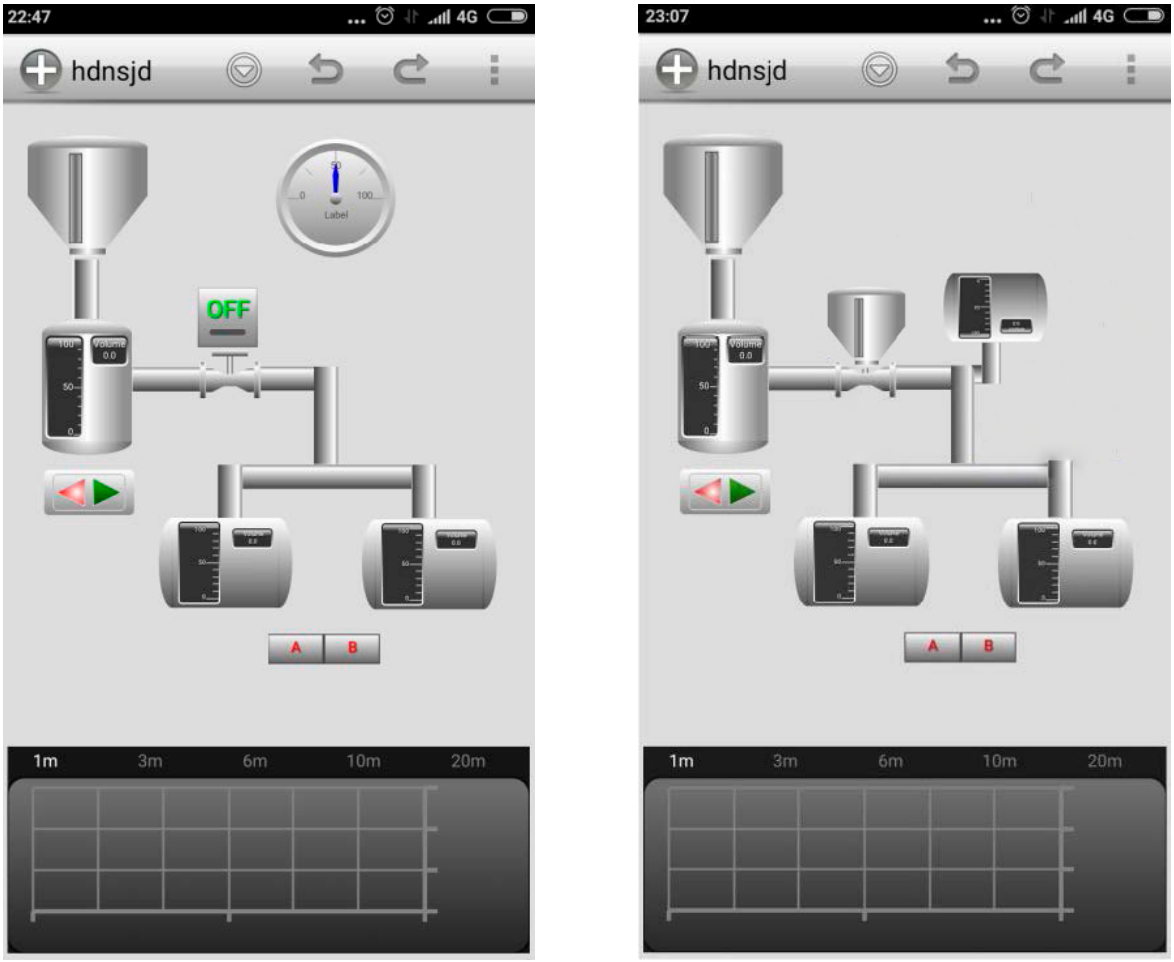

Figure 13: Proof-of-concept Attack on Victim HMI Panel View

Another problem that appeared in several client implementations was the incorrect use of Web Components. One of the analyzed applications was based on Apache Cordova and allowed JavaScript code to access several Android features. This application also loaded multiple HTML templates and CSS stylesheets from a remote server, using cleartext HTTP connections. As such, an attacker in a MiTM position could inject malicious JavaScript code in the application that will be able to execute with certain privileges and access to some device features (e.g., Bluetooth).

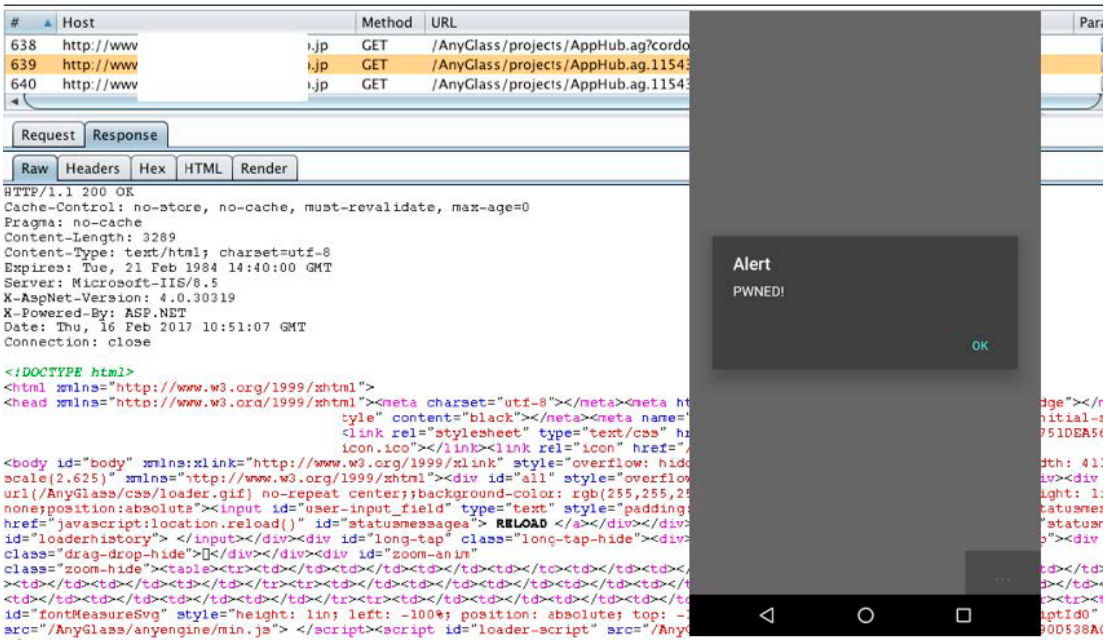

Figure 14: Client-side injection in HMI Panel Using Cordova

More than a third of the analyzed applications had one or more problems with client code (lack of) quality.

M8: Code Tampering

This category covers binary patching, local resource modification, method hooking, method swizzling, and dynamic memory modification.

Almost all of the analyzed applications (94%) did not implement code anti-tampering protections. Therefore, we only focused on reviewing the most basic anti-tampering protection found in mobile applications: rooted device detection. Rooted Android devices are relatively common, and they allow the user to perform actions such as installing custom firmware, removing system applications, or disabling system components. A drawback of rooting is that, for some devices, any regular application has the privileges to perform any of these actions as well, without user interaction. For instance, any installed application would be able to access the private folders of another sensitive application. We observed that running on a rooted device eases the exploitability of certain application vulnerabilities.

We acknowledge that any root detection mechanism implemented by a non-privileged application can fundamentally be evaded by a root-privileged malicious application; nevertheless, as with obfuscation, implementation of simple root detection mechanisms may raise the bar for an attacker or a malicious application.

Rooted-device detection can be done in many ways: for example, by checking for the presence of specific applications used to root devices, trying to access protected folders, checking the system settings, or attempting to set a new system property and read it back.

Best practice recommends that, at a minimum, the application implement several simple detection methods and show a pop-up warning if it detects it is running on a rooted device. Of the 34 applications we analyzed, only two implemented a basic root detection mechanism.

M9: Reverse Engineering

This category includes analysis of the final core binary to determine its source code, libraries, algorithms, and other assets.

We began by testing whether code obfuscation is in place. Code obfuscation could prevent an attacker from understanding program algorithms, discovering intellectual property, or extracting sensitive data. Also, bug exploitation by application patching can be made more complex, because it is harder for an attacker to determine the actual code blocks that need to be patched. Several obfuscation tools are available that integrate with development environments. A basic, yet effective, obfuscation approach consists of renaming all variables and functions, so that much of their semantic purpose is lost. As an example, the snippet below presents obfuscated code obtained by a Java decompiler:

```
8  public class bq
9    extends bm
10{
11   public bq(bh parambh, byte[] paramArrayOfByte)
12   {
13     super(parambh, paramArrayOfByte);
14   }
15
16   public dl a(cf paramcf)
17   {
18     dl localdl = dl.M;
19     ah localah = (ah)b(paramcf);
20     if (localah != null)
21     {
22       ed localed = (ed)this.b;
23       if (localed != null)
24       {
25         localah.a(localed.a, localed.b, localed.c, localed.g, localed.h);
26         return dl.a;
27       }
28     }
29     return localdl;
30   }
31
32   Object a(byte[] paramArrayOfByte)
33   {
34     dw localdw = p.d(bh.a(this.d));
```

Figure 15: Obfuscated Java Code

In the following example is non-obfuscated code obtained by a Java decompiler:

```
public class ProgressInputStream
  extends FilterInputStream
{
  protected long mTotal = 0L;

  public ProgressInputStream(InputStream paramInputStream)
  {
    super(paramInputStream);
  }

  public int read()
    throws IOException
  {
    byte[] arrayOfByte = { 0 };
    if (1 == read(arrayOfByte, 0, 1)) {
      return arrayOfByte[0];
    }
    return -1;
  }

  public int read(byte[] paramArrayOfByte, int paramInt1, int paramIn
    throws IndexOutOfBoundsException, IOException
  {
    int i = super.read(paramArrayOfByte, paramInt1, paramInt2);
    if (i > 0) {
      this.mTotal += i;
    }
```

Figure 16: Non-obfuscated Code

In general, non-obfuscated code is easier to understand, and it shows that even a simple approach of variable and function renaming could be an effective obfuscation technique.

However, less than half (47%) of the surveyed applications were implementing such techniques.

M10: Extraneous Functionality

Often, developers include hidden backdoor functionality or other internal development security controls that are not intended to be released into a production environment.

The main issue we identified in this category was related to application permissions. We found that a quarter (24%) of the applications appeared to be over-privileged and required permissions that were not used or did not seem to be related to the functionalities advertised by the applications.

Imagine two possible scenarios:

1. An application is compromised, and attackers can execute custom code from its process: extra permissions will give them an increased ability to pivot the attack
2. An application vendor is hacked or maliciously decides to include malware in the next application release: in this case, malicious code will be able to leverage extra permissions for harmful purposes

Those applications violated the principle of least privileges. In the following, we list permissions that could not be mapped to advertised or required application functionalities:

- android.permission.BLUETOOTH_ADMIN
- android.permission.GET_ACCOUNTS
- android.permission.USE_CREDENTIALS
- android.permission.MODIFY_AUDIO_SETTINGS
- android.permission.RECORD_AUDIO
- android.permission.ACCESS_COARSE_LOCATION
- android.permission.READ_LOG
- android.permission.RECEIVE_BOOT_COMPLETED
- android.permission.C2D_MESSAGE

One may argue that some applications (e.g., HMI panels) may need some of these permissions; for example, a local application may want to use coarse location to detect whether it is running inside the safe perimeter. However, using static and dynamic reverse engineering, we were not able to identify such additional functionalities. For example, we found a local HMI interface that has permission to record audio, but does not appear to have any audio-related functions, and a remote SCADA client that requires access to the user's location.

Backend Issues

While backend issues were not our main research target, we have included the vulnerabilities we identified while reviewing other application parts. The vulnerabilities we discovered involve remote backends and local servers, and include DoS, memory corruptions, SQL injections, and information leakage.

To our surprise, the most frequent backend vulnerability was SQL injection. Most of the reviewed SCADA backend applications that expose cloud endpoints are complex, modern web applications, sometimes storing data into databases. We observed that in many cases SQL input was not validated on the server-side; hence an attacker could trigger injections by maliciously using control symbols (e.g., quotes or comments) in the input parameters. See Figure 17 for a proof-of-concept affecting one of the surveyed applications. Additionally, one of the reviewed applications had write permissions for the tables, allowing an attacker to tamper with station configurations and user statistics.

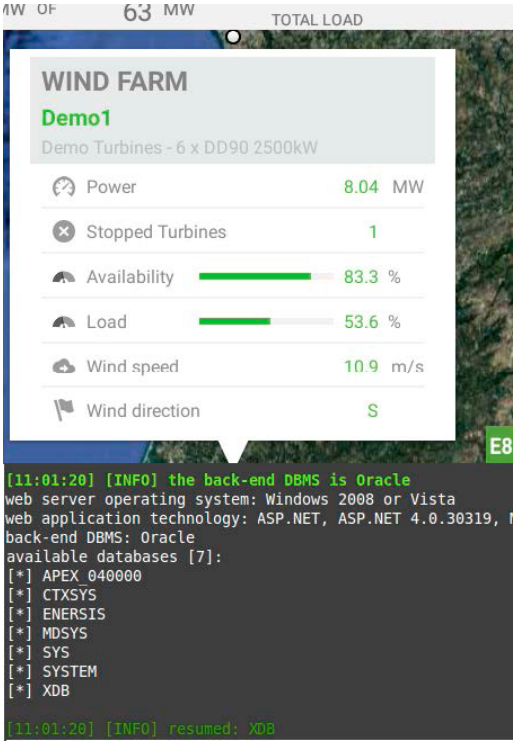

Figure 17: Example SQL Injection

Another common problem that mostly affected local applications (that is, those assumed to be running in the safe perimeter) was remote arbitrary file reading. Figure 18 shows a proof-of-concept where the server-side of an HMI mobile application allows arbitrary file reading. The problem lays in the incorrect processing of the supplied project path variable. An attacker, using path traversal, could read arbitrary files from the server. In one case, the local backend application was running with the privileged NT SYSTEM user by default, hence, the attacker could read sensitive systems files (e.g., SAM databases).

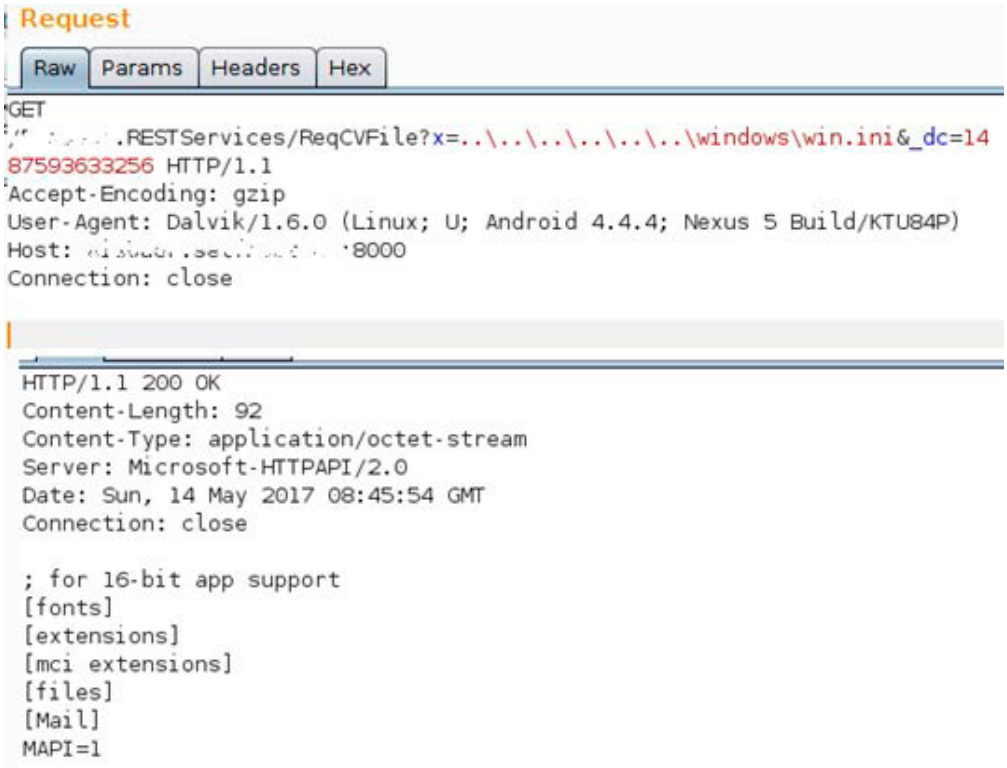

Figure 18: Arbitrary Remote File Read

The rest of the discovered backend issues are related to cross-site scripting (XSS), same-origin hijacking, and other web-related vulnerabilities. We also found DoS issues.

Remediation And Best Practices

In addition to the well-known recommendations covering the OWASP Top 10 and OWASP Mobile Top 10 2016 risks, there are several actions that could be taken by developers of mobile SCADA clients to further protect their applications and systems.

In the following list, we gathered the most important items to consider when developing a mobile SCADA application:

- Always keep in mind that your application is a gateway to your ICS systems. This should influence all of your design decisions, including how you handle the inputs you will accept from the application and, more generally, anything that you will accept and send to your ICS system.
- Avoid all situations that could leave the SCADA operators in the dark or provide them with misleading information, from silent application crashes to full subverting of HMI projects.
- Follow best practices. Consider covering the OWASP Top 10, OWASP Mobile Top 10 2016, and the 24 Deadly Sins of Software Security.
- Do not forget to implement unit and functional tests for your application and the backend servers, to cover at a minimum the basic security features, such as authentication and authorization requirements.
- Enforce password/PIN validation to protect against threats U1-3. In addition, avoid storing any credentials on the device using unsafe mechanisms (such as in cleartext) and leverage robust and safe storing mechanisms already provided by the Android platform.
- Do not store any sensitive data on SD cards or similar partitions without ACLs at all costs Such storage mediums cannot protect your sensitive data.
- Provide secrecy and integrity for all HMI project data. This can be achieved by using authenticated encryption and storing the encryption credentials in the secure Android storage, or by deriving the key securely, via a key derivation function (KDF), from the application password.
- Encrypt all communication using strong protocols, such as TLS 1.2 with elliptic curves key exchange and signatures and AEAD encryption schemes. Follow best practices, and keep updating your application as best practices evolve. Attacks always get better, and so should your application.
- Catch and handle exceptions carefully. If an error cannot be recovered, ensure the application notifies the user and quits gracefully. When logging exceptions, ensure no sensitive information is leaked to log files.
- If you are using Web Components in the application, think about preventing client-side injections (e.g., encrypt all communications, validate user input, etc.).
- Limit the permissions your application requires to the strict minimum.
- Implement obfuscation and anti-tampering protections in your application.

Conclusions

Two years have passed since our previous research, and things have continued to evolve. Unfortunately, they have not evolved with robust security in mind, and the landscape is less secure than ever before. In 2015 we found a total of 50 issues in the 20 applications we analyzed and in 2017 we found a staggering 147 issues in the 34 applications we selected. This represents an average increase of 1.6 vulnerabilities per application.

We therefore conclude that the growth of IoT in the era of "everything is connected" has not led to improved security for mobile SCADA applications. According to our results, more than 20% of the discovered issues allow attackers to directly misinform operators and/or directly/indirectly influence the industrial process.

In 2015, we wrote:

> *SCADA and ICS come to the mobile world recently, but bring old approaches and weaknesses. Hopefully, due to the rapidly developing nature of mobile software, all these problems will soon be gone.*

We now concede that we were too optimistic and acknowledge that our previous statement was wrong.

Over the past few years, the number of incidents in SCADA systems has increased and the systems become more interesting for attackers every year. Furthermore, widespread implementation of the IoT/IIoT connects more and more mobile devices to ICS networks. Thus, the industry should start to pay attention to the security posture of its SCADA mobile applications, before it is too late.

Acknowledgments

Many thanks to Dmitriy Evdokimov, Gabriel Gonzalez, Pau Oliva, Alfredo Pironti, Ruben Santamarta, and Tao Sauvage for their help during our work on this research.

About Us

Alexander Bolshev

Alexander Bolshev is a Security Consultant for IOActive. He holds a Ph.D. in computer security and works as an assistant professor at Saint-Petersburg State Electrotechnical University. His research interests lie in distributed systems, as well as mobile, hardware, and industrial protocol security. He is the author of several white papers on topics of heuristic intrusion detection methods, Server Side Request Forgery attacks, OLAP systems, and ICS security. He is a frequent presenter at security conferences around the world, including Black Hat USA/EU/UK, ZeroNights, t2.fi, CONFIdence, and S4.

Ivan Yushkevich

Ivan is the information security auditor at Embedi (http://embedi.com). His main area of interest is source code analysis for applications ranging from simple websites to enterprise software. He has vast experience in banking systems and web application penetration testing.

IOActive

IOActive is a comprehensive, high-end information security services firm with a long and established pedigree in delivering elite security services to its customers. Our world-renowned consulting and research teams deliver a portfolio of specialist security services ranging from penetration testing and application code assessment through to semiconductor reverse engineering. Global 500 companies across every industry continue to trust IOActive with their most critical and sensitive security issues. Founded in 1998, IOActive is headquartered in Seattle, USA, with global operations through the Americas, EMEA and Asia Pac regions. Visit for more information. Read the IOActive Labs Research Blog. Follow IOActive on Twitter.

Embedi

Embedi expertise is backed up by extensive experience in security of embedded devices, with special emphasis on attack and exploit prevention. Years of research are the genesis of the software solutions created. Embedi developed a wide range of security products for various types of embedded/smart devices used in different fields of life and industry such as: wearables, smart home, retail environments, automotive, smart buildings, ICS, smart cities, and others. Embedi is headquartered in Berkeley, USA. Visit for more information and follow Embedi on Twitter.

