Architecture-Driven Penetration Testing against an Identity Access Management (IAM) System

Sam Chung
Southern Illinois University
1365 Douglas Dr. Mailcode6614
Carbondale, IL 62901
1-618-453-7279
samchung@siu.edu

Sky Moon
Expedia
333 108th Ave NE #300
Bellevue, WA 98004
1-425-890-0163
cmoon@expedia.com

Barbara Endicott-Popovsky
Univ. of Washington Bothell
18115 Campus Way NE
Bothell, WA 98011-8246
1-206-284-6123
endicott@uw.edu

ABSTRACT
The purpose of this research is to propose architecture-driven, penetration testing equipped with a software reverse and forward engineering process. Although the importance of architectural risk analysis has been emphasized in software security, no methodology is shown to answer how to discover the architecture and abuse cases of a given insecure legacy system and how to modernize it to a secure target system. For this purpose, we propose an architecture-driven penetration testing methodology: 4+1 architectural views of the given insecure legacy system, documented to discover program paths for vulnerabilities through a reverse engineering process. Then, vulnerabilities are identified by using the discovered architecture abuse cases and countermeasures are proposed on identified vulnerabilities. As a case study, a telecommunication company's Identity Access Management (IAM) system is used for discovering its software architecture, identifying the vulnerabilities of its architecture, and providing possible countermeasures. Our empirical results show that functional suggestions would be relatively easier to follow up and less time-consuming work to fix; however, architectural suggestions would be more complicated to follow up, even though it would guarantee better security and take full advantage of OAuth 2.0 supporting communities.

CCS Concepts
• Security and privacy → Software and application security
• Information systems → Information systems applications.

Keywords
Software Architecture; Penetration Testing; Identity and Access Management Environment; OAuth 2.0;

1. INTRODUCTION
An Identity Access Management (IAM) system means "a framework for business processes that facilitates the management of electronic identities" [10]. The IAM will be necessary in the future for managing data security of Bring-Your-Own-Device (BYOD) or Cloud Computing [5, 10].

RIIT'16, September 28-October 01 2016, Boston, MA, USA
 ISBN 978-1-4503-4453-1/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2978178.2978183

In this paper, instead of blindly testing security functionality with standard functional testing techniques, we focus on improving software architecture to make software attacks difficult. Verdon and McGraw suggest that since one half of all security problems come from design flaws, performing a risk analysis at the design level is important [13]. For this reason, architectural risk analysis has been emphasized in software security to discover software design flaws and abuse cases based upon those flaws [1, 7, 9, 12]. However, since software security means the protection of software after it has been built and deployed, we encounter challenges: How can we discover architectural design and abuse cases from a deployed IAM system? Based upon the architecture and abuse cases, how can we identify vulnerabilities and propose countermeasures?

To answer these challenges, we consider a case study: a telecommunication company in the State of Washington had a plan to discover vulnerabilities of their IAM system before it launched. This plan raised an important question: how can vulnerabilities of the newly developed IAM system be identified and related vulnerabilities be mitigated? We then propose architecture-driven penetration testing using a reverse and forward software engineering process. During the reverse software engineering process, we first specify, document, and visualize the physical/logical and static/dynamic properties of a given insecure IAM architecture. In the forward software engineering process, the penetration testing team identifies vulnerabilities from a high level abstraction of the IAM source code, i.e. the visual model. Then, the team analyzes risks to the IAM and provides possible countermeasures, to make a secure target system.

2. HACKER'S PERSPECTIVE
Compared to software testing, penetration testing needs to find the absence of an unspecified behavior of a given insecure legacy system [12]. Therefore, thinking like an attacker is a must [9]. IAM hackers have exceptional knowledge and skills with networks, OS and web applications and programming languages. They are also assumed to know about the OAuth 2.0 specification - RFC 6749 and 6819 [8]. It is expected that hackers would want to get intact source code of applications running on the device which communicate with servers in order to analyze internal vulnerabilities of the application. Based on this assumed hacker behavior mentioned above, we primarily focused on the IAM system's access token because we assume that the IAM would implement the access token mentioned in OAuth 2.0 (RFC 6749) and that it is a key component of accessing user assets.

3. PREVIOUS WORK
Although the importance of architectural risk analysis has been proposed a decade ago [7, 9], based on an exhaustive search, only

a few articles were found that discuss using architecture for penetration testing. Those articles found focus on using architecture for risk analysis, as opposed to discovering the architecture of a given insecure legacy system. Recently, Xiong and Peyton used an AJAX web application architecture to propose a web application penetration test security model with three entities – application footprint, entry point, and check point [14]. Shin et al. used an architecture of their reactor protection system to analyze architectural risk [11]. Chung, et al., used a software re-documentation methodology called 5W1H Re-Doc to identify web services from a given legacy system through reverse software engineering process [3, 4]; however, this methodology was not applied to architectural risk analysis.

4. ARCHITECTURE-DRIVEN PENETRATION TESTING

Penetration test implies a test conducted from a hacker's perspective with approval from the test requesters. For this reason, we are interested in misuse cases, in addition to normal use cases. Since there is usually no information available at the start, we collect all necessary information to discover misuse cases through a reverse engineering process. We borrow the definitions of reverse and forward engineering from Chikofsky and Cross [2].

We propose an architecture-driven, penetration testing methodology to reengineer an insecure legacy system to a secure target system by discovering use cases for normal users and abuse cases for hackers through a reverse engineering process which identifies vulnerabilities based upon the abuse cases, and proposes countermeasures that will be used through a forward engineering process. Architecture-driven means the architecture of a given insecure legacy system will be the main information for penetration testing. Through the reverse engineering process, architecture of the legacy system is re-documented into a visual model that explains physical/logical and static/dynamic properties of the system.

An architecture re-documentation methodology called 5W1H Re-Doc [3, 4] using multiple views called 4+1 views [6] is applied to the IAM system and its architecture is described in a visual model using Unified Modeling Language (UML). From the visual model, we discover abuse cases that a hacker could exploit. A Computer-Aided Software Engineering (CASE) tool, Sparx's Enterprise Architect, is used to generate the visual model derived from reverse engineering the IAM system. Documentation of the architecture is provided for each component of the IAM system included, but not limited to Web Authentication Service, Token Service, Profile Service, Android Agent, and Helper Library. Then, we backtrack the discovered abuse cases and identify vulnerabilities. For identified vulnerabilities, we propose countermeasures based upon industry recommendations or practices. In our case study, we used OAuth 2.0 recommendations for an IAM development (OAuth, 2016).

5. SOURCE CODE RE-GENERATION

From the hacker perspective, we attempt to obtain the application's source code from the deployed legacy Android app for the IAM by following the four steps below:

Step 1 - Locate the APK file: we start with a given Android Application Package (APK) file.

1) Install Android Software Development Kit (SDK), available at http://developer.android.com/sdk/index.html.
2) Connect the given android device to a computer and type the 'adb' shell command.
3) Move to the Android app directory by typing 'cd /data/app.'
4) Type 'ls' to see the list of files located at the '/data/app' directory. From there, locate the target APK file.

Step 2 - Pull the APK file: Figure 1 shows the list of files located at '/data/app' directory and pulls the APK files by using the 'adb pull' command. On command prompt, type 'adb pull data/app/{APK filename}.apk {destination to save}.'

Figure 1. The list of '.apk' files and the 'adb pull' command

Step 3 - Convert the APK file to a Java Archive file (JAR) file\; however, we cannot yet read code from the pulled APK file. The APK file must be converted to a JAR.

1) Install a program called 'dex2jar,' available at https://code.google.com/p/dex2jar/.
2) Unzip the 'dex2jar' and put the APK file to where the 'dex2jar' is unzipped.
3) In the directory of the 'dex2jar', open up a terminal and type the following command: 'dex2jar {APK filename}.apk'

Step 4 - Open JAR file: Figure 2 shows the decompiled source code that was reverse engineered from the deployed APK file.

1) Install a program called 'JD-GUI,' available at http://jd.benow.ca/.
2) Run the 'JD-GUI' program and open the converted JAR file with 'JD-GUI.'

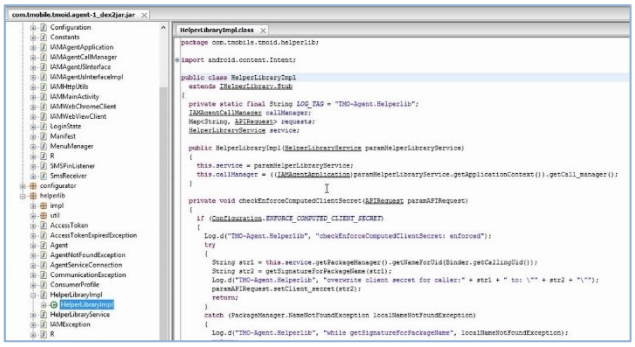

Figure 2. The decompiled JAR source file

6. ARCHITECTURE AND ABUSE CASES

Because there is no documentation on the decompiled source code, and variable names in the source code are automatically generated, it is very hard for a penetration tester to understand designs from the code directly. Therefore, we create a visual model that shows a high level abstraction of the given system. The Enterprise Architecture 10.0 CASE tool is used to document a visual architectural model.

Step 1 – Deployment View: the deployment view shows how system level processes are distributed and networked on which physical nodes. Because we are assuming the hacker perspective,

we do not know the system process of the target application yet; however, as we analyze the decompiled source code, we can understand the system, and fill out a deployment view as we progress. To set up a deployment view, we create a deployment diagram. In this diagram, we create a node with an Android device name, then add a component representing the APK file to the node. As we discover more nodes and their components, we add them to the deployment diagram, which is shown in Figure 3. The first time, the deployment view may be simple, but eventually it gains more components and nodes as the target application interacts with multiple nodes.

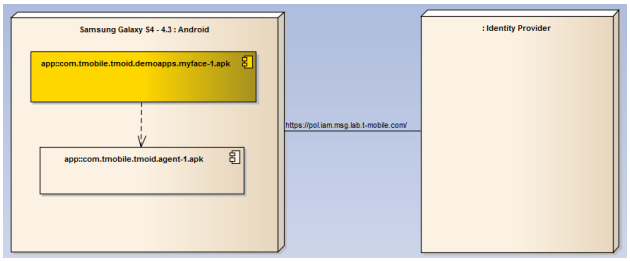

Figure 3. A deployment diagram

Step 2 - Component View: the component view, also known as an implementation view, shows location for the physical files and directories of the decompiled source code. To create a component view, we add packages for the discovered source directories. For each physical file in a directory, we place components under the directory. To show relationship among components or packages, we create component diagrams by using the components under packages.

Step 3 – Design View: the design view shows class level relationships. To generate a design view, we use a semi-automated approach. Instead of generating classes manually, we use the semi-automatic feature of the given Enterprise CASE tool: select the Design View (Class View) on the model, right click on Design View, select Code Engineering, and click Import Source Directory. Define the Root Directory and Source Type, and click OK. After all sources are imported, you still must manually manage the relationships between packages and verify the relationships between classes within each package.

Step 4 - Discover Process View: the process view, also known as a sequence view, shows internal message exchanges between objects. This view helps a penetration tester understand what methods were used and how they were called. A sequence diagram is shown in Figure 4. Select the Process View (Sequence View) on Model, add actors who trigger events, drag classes from the Design View that are being used, connect actors and classes in order of sequence being called, and define each connection name by choosing a method invoked.

Step 5 - Discover Use Case View: the use case view directory shows how a user can use the target mobile app. A user requirement is visualized under the use case view directory by using a use case diagram (Figure 5). A use case diagram consists of actors, a system(s) for the actors, a set of use cases within the system, and interactions between the system and actors, and relationships among use cases. We are interested not only in use cases for a normal user, but also in abuse cases for a hacker. Therefore, we run the actual mobile app to determine both use and abuse cases for Alice (user) and Eve (hacker) in Figure 5, while we are tracing the discovered sequence diagram of Figure 4.

7. VULNERABILITIES

With the discovered abuse cases and the architecture in the UML visual model, a hacker can understand options for abusing the system as well as design weaknesses of the legacy mobile app at a high level of abstraction. From the hacker's perspective, we can view different levels of each of 4+1 view. From the Use Case View, we notice that the mobile app fetches profile information without authentication. Due to automatic fetches of information, some profile information must be loaded. We then use the Process View to look at mobile app message exchanges and found that an access token is being saved into a physical memory space called 'SharedPreference.'

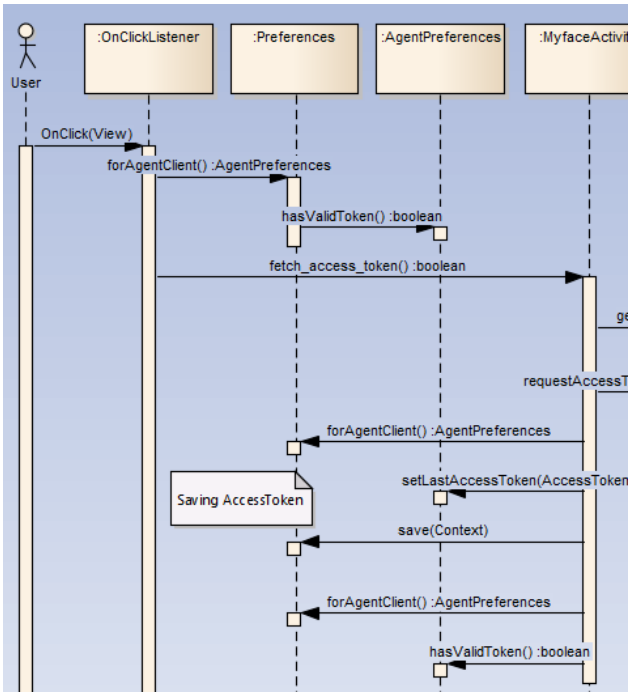

Figure 4. A sequence diagram

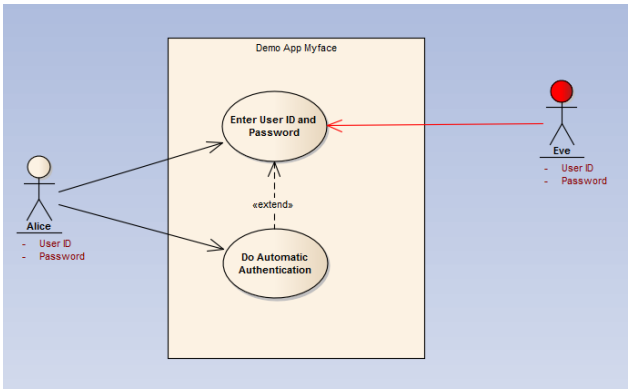

Figure 5. A use case diagram for a user and a hacker

By using the information in 'SharedPreference,' we attempt to discover information on server endpoints and an access token. We perform experiments using Android smartphones and one virtual device run by Genymotion (http://www.genymotion.com/). The following devices are tested: Samsung Galaxy S3 (Android 4.3), LG P659 (Android 4.2), LG G2 (Android 4.2.2), Genymotion 2.1.1 virtual device Samsung Galaxy S4 - 4.3, Android Virtual Device 4.1.2 API 16. In order to discover resource server addresses, we

identify the URL of login, profile, authorization, token, and redirect from the Use Case View, and find 'SharedPreference' used for saving this information. We locate the actual file using the following steps:

1) Connect the android device with a workstation. Then, type 'adb' shell at command prompt or terminal.
2) Go into directory '/data/data' by using 'cd' command. With 'ls' command, list directories and files of the APK files of the given mobile app.
3) Find a file called 'shared_prefs' and move to its directory. There should be a file called 'agent_preferences.xml' file.
4) To pull the file from the Android device, type 'adb pull' at command prompt or terminal with the xml file and its location to be stored. In Figure 6, you can find several URLs to servers.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="device.agent">IAM_Device_Agent/0.5.6</string>
<string name="profile.url">https://pol.iam.msg.lab.t-
mobile.com/consumerProfile/v1</string>
<boolean name="enforce_computed_client_secret" value="false" />
<string name="redirect.uri">https://localhost</string>
<string name="accesstoken.url">https://pol.iam.msg.lab.t-
mobile.com/oauth2/v1/token</string>
<string name="dashboard.url">https://pol.portal.iam.msg.lab.t-
mobile.com/primary/dashboardPage</string>
<boolean name="clearcache" value="false" />
<string name="authorize.url">https://pol.portal.iam.msg.lab.t-
mobile.com/primary/Oauth</string>
</map>
```

Figure 6. Information of server endpoints used an XML file

In order to discover an access token, similar steps to discover resource server addresses are used. We discover the access token file in an XML file called 'myface_preferences.xml,' which is located under 'shared_prefs' (Figure 7).

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="access_token.token_type">Bearer</string>
    <long name="access_token.create_time" value="1398901987881"
/>
    <int name="access_token.expires_in" value="3600" />
    <string name="access_token.scope">TMO_ID_profile</string>
    <string name="access_token.access_token">
R9QC19FmrCzqSR5cv1NW</string>
    <string
name="access_token.tmobileid">2a2ff76c0b80460b8af6d4d73f1d30e2345
ae4166e8cfb39d3d8b35a54c3af7d</string>
    <null name="access_token.refresh_token" />
</map>
```

Figure 7. Information of access token in an XML file

Next, by using the server end-points (authorization and resource server) and the access token, we can retrieve a user profile. Three approaches are attempted: cURL, Python script, and Android App. All three approaches show that we can successfully retrieve a user profile.

Approach 1 – cURL: We first get the user profile with a command called cURL. The *cURL* is a Linux command to manually construct a HTTP request and send to a URI. By using the cURL command, 'curl --insecure --header "Authorization: Bearer {access token}" {resource server URI},' we successfully retrieve a user profile using the access token.

Approach 2 – Python Script: we retrieve the user profile with a Python script. The Python version for this script is version 2.7.6 (This can be downloaded here https://www.python.org/ under the download tab). The first thing that we need to do is to install two modules: 1) Requests module at http://docs.python-requests.org/en/latest/user/install/#install and 2) OAuth2Session module at https://github.com/requests/requests-oauthlib. To confirm the installation, run Python (command line) and type in 'import requests' then press *Enter*. If it only creates a new line to enter information, then it has been correctly installed. Do the same with the other module by typing 'from requests_oauthlib import OAuth2Session' and then press *Enter*. The Python script in Figure 8 is used to get the user's profile.

```
#Importing Modules
import requests
from requests_oauthlib import OAuth2Session
raw_input('Press any key to execute')
#OAuth access token
token = {
 'access_token': '', #Access token goes in single quotes
 'token_type': 'Bearer'
}
#Setting access token and URL
TMO = OAuth2Session(r'client_id', token=token)
url='https://pol.iam.msg.lab.t-
mobile.com/userprofile/p/v1/userinfo'
#Sending token
r = TMO.get(url, verify=False) #verify=False is need if the cert
is not valid
#Display response information
print '~~~HTTP Status report~~~'
print r.status_code
print ''
print '~~~Header~~~'
print r.headers
print ''
print '~~~Content~~~'
print r.content
print '
raw_input('Press any key to continue...')
```

Figure 8. A Python script 'getUserInfo.py'

Along with this Python script, there is another script that can be used to generate every possible access token combination.

```
#You'll need to import these modules before running the script
import itertools, string, sys
#Creating list of a possible token
#repeat is the length of each possible string
try_token=map(''.join,itertools.product(string.ascii_uppercase
+ string.ascii_lowercase + string.digits, repeat=1))
print try_token
```

Figure 9. A Python script 'genToken.py'

With this Python script approach, there is a way to perform an attack on the system that can affect multiple users. The scenario is as followed: a hacker logs onto the system with his or her own Android device. The hacker then checks the logs of the Android device and discovers the URL location of the resource server and its access token. After marking this information down, the hacker tries again to see how the access token changes. The hacker discovers a pattern with the access token and marks down the length, characters and numbers that recur. The hacker creates a Python script (e.g. genToken.py in Figure 9) to create all the possible access token combinations and then eliminates any access

token combination that does not fit to the recurring access tokens (e.g. AAAAA). After creating a word list of all possible access tokens, the hacker then creates a Python script that sends a request to the resource server (e.g. getUserInfo.py in Figure 8). Furthermore, with Python, the hacker can add a loop command to go through each possible access token without having to manually change the variable value. By performing this step, a hacker has the ability to get multiple users' information within the time frame available.

Approach 3 – Android App: we retrieve a user profile with Android App. An 'AgentHack' application uses 'iam-helper' to interact with 'iam-agent.' The application is built on Android SDK 4.4 (http://developer.android.com/sdk/index.html). Because the application depends on 'iam-agent,' the application does not need to know the server URL. The application creates the 'AccessToken' object with 20 character-long string input, sends it to 'iam-agent', and then receives user profile information (Figure 10).

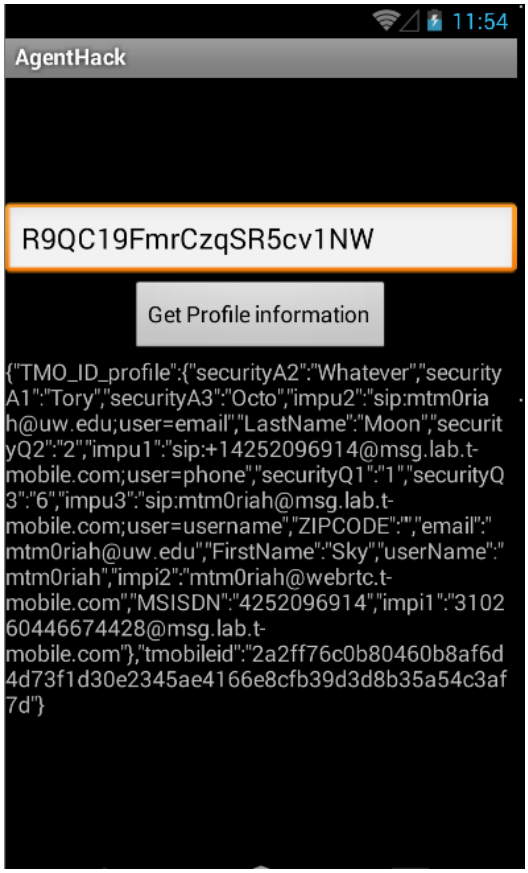

Figure 10. A fetched user profile by using the 'AgentHack' Application

8. COUNTERMEASURES

We successfully obtained a user profile from the resource server using the access token extracted from the Android file system. For each identified vulnerability for Android app and server endpoints, we recommend two reliable countermeasures, with references to RFC 6819, for the Android app and server endpoint vulnerabilities, respectively (OAuth, 2016).

The following countermeasures are proposed for the Android app vulnerabilities:

- Do not log the access token retrieval part (RFC6819 Section 4.6.7). Accidently, developers of the 'iam-helper' library did not remove the logs for the access token retrieval.
- Use Authorization headers or POST parameters instead of URI request parameters (RFC 6819 Section 5.4.1) - "Authorization headers are recognized and specially treated by HTTP proxies and servers. Thus, the usage of such headers for sending access tokens to resource servers reduces the likelihood of leakage or unintended storage of authenticated requests in general, and especially Authorization headers."
- Keep the access token in transient memory and limit grants (RFC6819 Section 5.1.6). The access token should not be stored in a physical file system. There may be a way to get data even from transient memory, but it would be much more difficult.
- Keep the access token in private memory or apply the same protection means as for refresh tokens (RFC6819 Section 5.2.2). We also need to store the refresh token in private memory for the refresh token. Do not store it in a physical file system.
- Limit the access token's scope (RFC6819 Section 5.1.5.1). It is better to limit the privilege of the access token, if you implemented the privilege mechanism.
- Keep the access token's lifetime short (RFC6819 Section 5.1.5.3.) The shorter the lifetime, the more secure your system. Currently the lifetime is one hour.

A countermeasure proposed for the server endpoints vulnerability follows:

- Insert a blocking mechanism (i.e., blocking a resource request from the same IP address, if it fails more than 3 times within a time interval) to prevent a brute-force attack.

9. CONCLUSION & FUTURE WORK

In order to discover architectural design and abuse cases from a deployed insecure legacy system, we borrowed ideas from software reengineering: we consider a given system as a legacy system that may have security vulnerabilities, reverse engineer the given legacy system to identify possible vulnerabilities, and then propose countermeasures for a target system that won't have those vulnerabilities. We apply a reverse engineering methodology called 5W1H Re-Doc to a given legacy system and discover the system architecture from the hacker's view.

In our case study of a tele-communication community, we discover that we can retrieve the server endpoints and an access token that are insecurely stored in Android common storage by backtracking an abuse case from a generated visual model. With the retrieved information, we demonstrate that we can obtain a user profile from a server by using three different approaches – a tool, script programming, and a mobile device. Then, we propose countermeasures for the Android app and the server endpoints with references to OAuth RFC 6819.

This case study suggests a promising future for architecture-driven penetration testing to help a security engineer identify vulnerabilities from nothing (black-box penetration testing) to architecture (white-box penetration testing) and prepare for countermeasures against identified vulnerabilities by considering both physical and cyber properties with multiple and hierarchical architectural views. More case studies are planned in the near future to research including diverse Cyber-Physical Systems (CPSs) in preparation for the Internet of Things (IoT).

10. REFERENCES

[1] Arkin B., Stender, S., and McGraw, G. 2005. Software Penetration Testing, *IEEE Security & Privacy*, 3, 1, (Mar. 2005), 84-87. DOI = 10.1109/MSP.2005.23

[2] Chikofsky, E. J., and Cross, H. J. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7, 1 (Jan. 1990), 13-17. DOI = 10.1109/52.43044

[3] Chung, S., Crompton, C., Bai, Y., Endicott-Popovsky, B., and Park, S. 2012. *Analyses of Evolving Legacy Software into Secure Service-Oriented Software using Scrum and a Visual Model*. Agile and Lean Service-Oriented Development: Foundations, Theory, and Practice. Edited by Wang, X., Ali, N., Ramos, I., & Vidgen, R., Information Science Reference, Hershey, PA.

[4] Chung, S., Won, D., Baeg, S., and Park, S. 2009. Service-Oriented Reverse Reengineering: 5W1H Model Driven Re-documentation and Candidate Services Identification. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications* (Taipei, Taiwan, Jan. 14-15, 2009). SOCA '09. IEEE, Piscataway, NJ. DOI = 10.1109/SOCA.2009.5410445

[5] Cser A. and Maxim, M. 2016. IAM is the future for managing data security, (Mar. 2016), ComputerWeekly.com, http://www.computerweekly.com/feature/IAM-is-the-future-for-managing-data-security, retrieved June 6, 2016.

[6] Kruchten, P. B. 1995. The 4+1 View Model of Architecture. *IEEE Software*, 12, 6 (Nov. 1995), 42-50. DOI = 10.1109/52.469759

[7] McGraw, Software Security. *IEEE Security & Privacy*, 2, 2 (Apr. 2004), 80-83. DOI = 10.1109/MSECP.2004.1281254

[8] OAuth Documentation. https://tools.ietf.org/html/rfc6749, retrieved June 6, 2016.

[9] Potter, B., and McGraw, G. 2004. Software Security Testing, *IEEE Security & Privacy*, 2, 5 (Oct. 2004), 81-85. DOI = 10.1109/MSP.2004.84

[10] Rouse, M. 2015. Identity Access Management (IAM) System. http://searchsecurity.techtarget.com/definition/identity-access-management-IAM-system, retrieved June 6, 2016.

[11] Shin, J., Son, H., and Heo G. 2013. Cyber Security Risk Analysis Model Composed with Activity-quality and Architecture Model. In *Proceedings of International Conference on Computer, Networks and Communication Engineering* (2013). ICCNCE '13. 609-612. Atlantis Press.

[12] Thomson, H. H. 2005. Application Penetration Testing, *IEEE Security & Privacy*, 3, 1 (Feb. 2005), 66-69. DOI = 10.1109/MSP.2005.3

[13] Verdon, D., and McGraw, G. 2004. Risk Analysis in Software Design. *IEEE Security & Privacy*, 2, 4 (Aug. 2004), 32-37. DOI = 10.1109/MSP.2004.55

[14] Xiong, P. and Peyton, L. 2010. A Model-Driven Penetration Testing Framework for Web Applications. In *Proceedings of the 2010 Eighth International Conference on Privacy, Security and Trust* (Ottawa, Canada, Aug. 17-19, 2010). PST '10. 173-180, IEEE, Piscataway, NJ. DOI = 10.1109/PST.2010.5593250