Docker Notary

Application Penetration Test

Prepared for:

Prepared by:

Tom Ritter

Aleks Kircanski

Tyler Curtis

Table of Contents

1 Executive Summary

Application Summary	
Application Name	Notary
Application Version	1.0
Application Type	Go

Engagement Summary	
Dates	July 27, 2015 – July 31, 2015
Consultants Engaged	3
Total Engagement Effort	3 person-weeks
Engagement Type	Application Penetration Test
Testing Methodology	White Box

Vulnerability Summary	
Total High severity issues	2
Total Medium severity issues	0
Total Low severity issues	4
Total Informational severity issues	2

Total vulnerabilities identified: 8

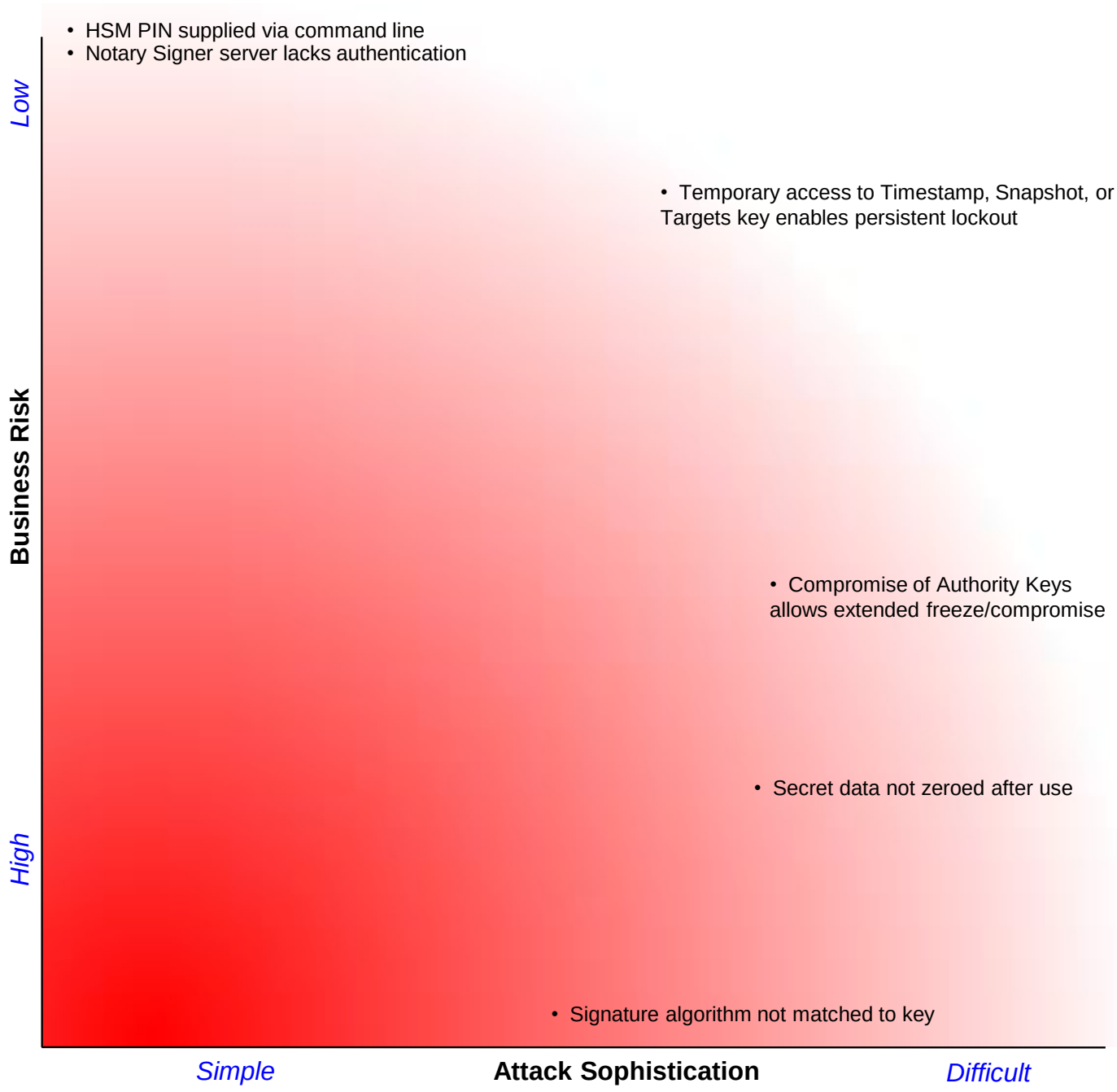
See section 3.1 on page 10 for descriptions of these classifications.

Category Breakdown:

Access Controls	0
Auditing and Logging	0
Authentication	1
Configuration	1
Cryptography	1
Data Exposure	3
Data Validation	0
Denial of Service	1
Error Reporting	1
Patching	0
Session Management	0
Timing	0

1.1 Cryptography Services Risk Summary

The Cryptography Services (CS) Risk Summary chart evaluates vulnerabilities according to business risk. The impact of the vulnerability increases towards the bottom of the chart. The sophistication required for an attacker to find and exploit the flaw decreases towards the left of the chart. The closer a vulnerability is to the chart origin, the greater the business risk.



1.2 Project Summary

Docker engaged NCC Group to perform a review of the Notary system. Notary is a new, fully open-source solution for signing Docker images. Notary allows publishers to manage and sign their own images, using keys that they manage. NCC Group's Cryptography Services ("CS") reviewed the security of Notary's cryptographic and application aspects. The engagement began on July 27, 2015 and concluded on July 31, 2015. Three consultants worked on the project during one calendar week. In addition, one intern assisted the project. The following core items were looked into:

- Notary Client/Server
- Notary Signer
- The Update Framework's (TUF) use within Notary

The efforts CS provided during this review should not be considered a comprehensive review of the complete Notary system. Due to the complexity of the TUF library and its use within Notary, the time allotted did not allow an exhaustive review. The area of least review was the user's interaction with the `notary` client application, while the area that received the most was the `gotuf` library.

CS reviewed the Notary framework for TUF specific issues, general cryptographic concerns and potential vulnerabilities common for client/server applications. The review of `notary` and the `gotuf` library were primarily performed through source code review.

1.3 Findings Summary

In the `gotuf` library, the finding with the most impact was a signature confusion attack listed in finding 1 on page 13. This could allow an attacker to forge signatures by tricking a client into interpreting one encryption algorithm's key as another algorithm's key. For instance, if a RSA-PSS key is misinterpreted as an ed25519 key, the attacker may be able to perform the Elliptic Curve Discrete Log algorithm to recover the private key corresponding to this (weak) ed25519 key, and sign documents with it.

The other findings in the `gotuf` library relate to conditions that occur if one or more of the authority keys is compromised. In finding 3 on page 15, a scenario in which an attacker who compromises the Timestamp Authority key and signs a document with a version number that is excessively high is discussed. It is shown that such an attacker who has stolen an authority key can lock the user out from updating as the version number cannot be exceeded. A lack of expiry verification on the root keys is reported in finding 2 on page 14. This allows an attacker who manages to compromise the Timestamp, Snapshot and Target Keys to continue to operate the repository as per usual and add malicious files.

As shown in finding 4 on page 16, there is a lack of authentication in the Notary Signer service. An attacker who is able to access the Notary Signer network can perform arbitrary signatures with arbitrary keys stored in the database as well as remove and add new keys.

1.4 Recommendations Summary

Short Term

Short term recommendations are meant to be relatively easy actions to execute, such as configuration changes or file deletions that resolve security vulnerabilities. These may also include more difficult actions that should be taken immediately to resolve high-risk vulnerabilities. This area is a summary of short term recommendations; additional recommendations can be found in the vulnerabilities section.

Deprecate and document use of Prefix Hashes. Prefixes hashes may inadvertently open new folders up to control by unauthorized persons, if the folder added happens to contain a hash prefix that another role is authorized for. This model makes it extremely dangerous to use prefix hashes if folders are expected to be added to a hierarchy in the future.

Add authentication to the Notary Server. Even though the Notary Signer is meant to be behind firewalls and only accessible from the Notary Server host, different deployments of the Notary system may not respect this. Require strong authentication tokens when executing the Notary Signer API functions.

Long Term

Long term recommendations are more complex and systematic changes that should be taken to secure the system. These may include significant changes to the architecture or code and may therefore require in-depth planning, complex testing, significant development time, or changes to the user experience that require retraining.

Specify the maximum expiry time allowed for each key in the root document. This will limit the impact of a compromise of any individual key to this maximum expiry date.

2 Engagement Structure

2.1 Internal and External Teams

The Cryptography Services team has the following primary members:

- Tom Ritter — Security Consultant
- Aleks Kircanski — Security Consultant
- Tyler Curtis — Security Consultant
- Nik Kinkel — Security Consultant

The Docker team has the following primary members:

- Nathan McCauley — Docker
- Diogo Mónica — Docker
- David Lawrence — Docker

2.2 Project Goals and Scope

As part of the primary goals of this engagement, NCC Group

- Reviewed cryptographic solutions applied to ensure the integrity of signing in Notary
- Reviewed Notary's reliance on The Update Framework (TUF)
- Reviewed functional and design documentation describing the usage of encryption and verifiers considering both security and functional requirements customized to the particular deployment of these cryptographic implementations
- Analyzed the planned use cases and assurance desired of the system in comparison to the design documents and libraries in use
- Identified other common application vulnerabilities

In more detail, reviewing Notary's cryptographic solutions included verifying encryption/signing algorithms in use, use of secure randomness, key provisioning distribution and management, appropriate cryptographic algorithm parameters, protocol sequences and data flows.

While searching for common application vulnerabilities in client/server, NCC Group performed the following:

- Analyzed any applicable documentation
- Sent malicious data through the user interface and through direct connections to the application
- Attempted to bypass and/or exploit security weaknesses in the authentication and authorization mechanisms
- Searched for the ability to escalate privileges
- Identified security weaknesses that lead to access, unintended application usage, or loss of data integrity
- Performed necessary supplemental research and development activities to support analysis
- Identified and validated potential vulnerabilities
- Prioritized vulnerabilities based on ease of exploit, level of effort to remediate, and severity of impact if exploited
- Identified issues of immediate consequence and recommend solutions
- Developed long-term recommendations to enhance security
- Evaluated the deployment process

3 Detailed Findings

3.1 Classifications

The following section describes the classes, severities, and exploitation difficulty rating assigned to each issue that CS identified.

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users, and assessment of rights
Auditing and Logging	Related to auditing of actions, or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to mathematical protections for data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to the race conditions, locking, or order of operations

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small, or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, of moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation or serious legal implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

3.2 Vulnerabilities

The following table is a summary of vulnerabilities identified by CS. Subsequent pages of this report detail each of the vulnerabilities, along with short and long term remediation advice.

Vulnerability	Class	Severity
1. Signature Algorithm Not Matched to Key	Cryptography	High
2. Compromise of Authority Keys Allows Extended Freeze/Compromise	Data Exposure	High
3. Temporary Access to Timestamp, Snapshot, or Targets Key Enables Persistent Lockout	Denial of Service	Low
4. Notary Signer Server Lacks Authentication	Authentication	Low
5. Secret Data Not Zeroed After Use	Data Exposure	Low
6. HSM PIN supplied via command line	Configuration	Low
7. Named Collections Directory is World Readable	Data Exposure	Informational
8. Failure to Load Notary Server TLS Certificate Disables TLS	Error Reporting	Informational

3.3 Detailed Vulnerability List

1. Signature Algorithm Not Matched to Key

Class: Cryptography **Severity:** High **Difficulty:** Medium

Finding ID: CS-DRNT15-1

Targets:
`gotuf/signed/verify.go:VerifySignatures()`,
`gotuf/signed/verify.go:VerifyRoot()`

Description: In the `VerifySignatures` and `VerifyRoot` functions, after a key is retrieved from the database, the attacker-controlled `Method` attribute (containing the signature algorithm) is used to determine which signature verifier is called. Although the attacker cannot control the key, they may be able to forge a signature when the public key for one algorithm is interpreted as the public key for another. In the individual Key Verifier methods, the Public Key is not consistently checked to assert that it is of the correct type for this verification function.

This type of vulnerability is very similar to an attack on SSL 3.0 described by Wagner and Schneier,[1] where a public key of one algorithm is detected as a public key of another. This flaw was relived in 2012[2] and, although less directly related, again in 2015.[3]

Exploit Scenario: An attacker changes an RSA-PSS key to claim the ed25519 algorithm. The `Ed25519Verifier` does not check that the key is of the correct type, so an attacker interprets the RSA public key as an ed25519 public key, and is able to perform the Elliptic Curve Discrete Log problem to recover the private key. This ed25519 keypair is weak and insecure, but could be used to forge signatures within the context of `gotuf`.

Short Term Solution: Two steps should be taken to address this flaw:

1. The attacker-controlled signature method should not be used to look up the verification function, but rather the algorithm for the public key retrieved from the database.
2. Each verification function should perform a type-check to assert that it is being called with a key of the correct type.

Long Term Solution: Consider annotating or hiding the `Method` field on a Signature to ensure that it is not used in the future – the method should always be retrieved by the trusted public key stored in the database, rather than the attacker-controlled value.

Note: Docker resolved this issue during the engagement by adding the checks to the verification functions. The sig method is needed to determine which flavor of signing is expected for the same key type (RSA-PSS vs PKCS#1v1.5)

[1] https://www.schneier.com/paper-ssl.pdf Section 4.4
[2] http://www.cosic.esat.kuleuven.be/publications/article-2216.pdf
[3] https://weakdh.org/

2. Compromise of Authority Keys Allows Extended Freeze/Compromise

Class: Data Exposure **Severity:** High **Difficulty:** High

Finding ID: CS-DRNT15-2

Targets: `gotuf/client/client.go:checkRoot()`

Description: A `root.json` file may be used by `gotuf` after expiry, as long as other files refer to it. If an attacker compromises the other authority keys, they may choose to refer to an expired `root.json` file that still lists them as the correct keys. The compromise of the other keys cannot be overridden by the root key if the root document is never updated. The `checkRoot()`, which is in the critical path, contains a comment that the expiry will be checked, but this check is not actually performed.

```
// checkRoot determines if the hash, and size are still those reported
// in the snapshot file. It will also check the expiry, however, if the
// hash and size in snapshot are unchanged but the root file has expired,
// there is little expectation that the situation can be remedied.
func (c Client) checkRoot() error {
  role := data.RoleName("root")
  size := c.local.Snapshot.Signed.Meta[role].Length
  hashSha256 := c.local.Snapshot.Signed.Meta[role].Hashes["sha256"]

  raw, err := c.cache.GetMeta("root", size)
  if err != nil {
    return err
  }

  hash := sha256.Sum256(raw)
  if !bytes.Equal(hash[:], hashSha256) {
    return fmt.Errorf("Cached root sha256 did not match snapshot root sha256")
  }
  return nil
}
```

Listing 1: `checkRoot()`

Exploit Scenario: An attacker manages to compromise the Timestamp, Snapshot, and Target keys. The legitimate operator is able to observe the key compromise, and produces an updated `root.json` file to roll the keys, but the attacker produces malicious update files that refer to the old `root.json` document. The attacker is able to continue this process perpetually even after the `root.json` document expires. In order to remain a degree of stealth, the attacker could continue to operate the repository as per usual, and may even choose to track upstream development by supplying the user with updates to the software (which contain the attacker's newest update of their rootkit as well).

Short Term Solution: Add a check for the expiry of the root document to this function, and if it fails, re-download the root document.

Long Term Solution: Ensure that expiry checks are performed consistently on all keys and documents.

Note: In Docker's deployment, the `root.json` file expires at the same time as the root key, which does have it's expiration checked. Having a shorter-lived `root.json` file would enable rolling of the keys if they were compromised, which would not be feasible with a long-lived `root.json` file. To address the more generic use case, this check was added and confirmed before the completion of the engagement.

3. Temporary Access to Timestamp, Snapshot, or Targets Key Enables Persistent Lockout

Class: Denial of Service **Severity:** Low **Difficulty:** High

Finding ID: CS-DRNT15-3

Targets: `gotuf/signed/client.go:Update()`

Description: In the `Update()` function of `client.go`, if a timestamp, snapshot, or targets error occurs, the Root will be downloaded and the update process restarted. However this only occurs if certain errors occur – if others occur the entire update process aborts. While a network adversary may persistently disrupt network communication, TUF is designed to be resilient in the face of a transient network attacker. However, due to the retry mechanism, a temporary network attacker who also compromises an authority key is able to persistently lock clients out from updating.

If they specify a large version number, the document will be accepted and cached by clients. When the next, legitimate document comes in, it will be rejected with an `ErrLowVersion` error. This is an unrecoverable error, and `gotuf` will abort attempting to update without downloading a new Root. The only way to recover from this condition is to specify a document with the old (compromised) authority key and a higher version number (which may be unknown).

```
func (c *Client) Update() error {
        err := c.update()
        if err != nil {
                switch err.(type) {
                case signed.ErrRoleThreshold, signed.ErrExpired, tuf.
                    ErrLocalRootExpired:
                        logrus.Debug("retryable error occurred. Root will be
                            downloaded and another update attempted")
                        if err := c.downloadRoot(); err != nil {
                                logrus.Errorf("client Update (Root):", err)
                                return err
                        }
                default:
                        logrus.Error("an unexpected error occurred while updating TUF
                             client")
                        return err
                }
                logrus.Debug("retrying TUF client update")
                return c.update()
        }
        return nil
}
```

Listing 2: The `Update()` function, with the affected switch statement

Exploit Scenario: An attacker compromises the Timestamp Authority key, and signs a document with a version specifying `MaxInt32`, the greatest Integer that can be processed in golang by `gotuf`. This version number cannot be exceeded, and thus persistently fails, locking the user out from updating.

Short Term Solution: Add the `ErrLowVersion` error to the retry case. Additionally, when a new Authority key is received, it should allow a one-time reset to 0 of the current version required for that document.

Long Term Solution: It is likely that the switch should be eliminated entirely to prevent a recurrence of this vulnerability from occurring in the event of a different error.

Note: The switch statement was removed and confirmed after the completion of the engagement.

4. Notary Signer Server Lacks Authentication

Class: Authentication **Severity:** Low **Difficulty:** Low

Finding ID: CS-DRNT15-4

Targets: Notary Signer

Description: Notary Signer exposes an API that allows performing operations such as signing by chosen keys, creating/deleting new keys and obtaining info about keys already existing in the database. The service can be seen as a "crypto anchor" in the sense that it adds a level of indirection in the system and allows not keeping the signing keys on the Notary server. This is a good practice, and provided defense-in-depth; as the notary server becomes less privileged - but the minimal signing service should be appropriately secured.

The Notary Signer API currently does not implement any form of authentication, which allows anyone who has network access to the service to sign data with whatever keys are loaded into the signer. Even a rudimentary form of authentication would provide some defense-in-depth for this service, and it is not documented that it should be protected in some way.

Exploit Scenario: An organization wishes to be at the forefront of Docker security and integration, so its developers deploy Notary and Notary Signer without performing a thorough review of the system. An attacker who has compromised a work-from-home employee scans the internal network and finds the Notary Signer service, and after googling is able to deduce the type of service that is operating. They use it to sign malicious Docker images and masquerade as the organization, causing both the organization and Notary to have embarrassing early coverage in the media.

Short Term Solution: To allow for a basic level of protection, add options to run Notary Signer over HTTPS, and support HTTP Basic Authentication with a password specified in a config file. Document the mode of authentication, need to change or set the password, and suggest network-level access controls and other typical hardening in a real deployment.

Long Term Solution: If Notary Signer is intended to be run by individuals or organizations, improve the documentation to outline the important configuration changes necessary, the appropriate network or infrastructure security requirements, as well as maintenance and logging expectations.

Note: CS confirmed with the Development Team that adding authentication is planned in future releases. In the meantime, the service is expected to be firewalled off from the Internet and only accessible by the Notary Server. Due to these reasons, the severity of this finding is reduced to Low.

5. Secret Data Not Zeroed After Use

Class: Data Exposure **Severity:** Low **Difficulty:** High

Finding ID: CS-DRNT15-5

Targets: Notary functions where secret data such as private keys, decryption passphrases, and HSM pins are processed.

Description: If memory holding secret data is not explicitly overwritten before being garbage collected, the secret data may stay in memory and remain accessible for an unspecified period of time. Later memory allocations may return the data as uninitialized data.

Information such as private key material and encryption passphrases should be overwritten before the corresponding variables are allowed to go out of scope, and ideally as soon as the data is no longer needed for processing.

```
    case data.ECDSAKey:
        privKey, err = trustmanager.GenerateECDSAKey(rand.Reader)
    default:
        return "", fmt.Errorf("only RSA or ECDSA keys are currently supported. Found:
            %s", algorithm)
    }
    if err != nil {
        return "", fmt.Errorf("failed to generate private key: %v", err)
    }

    // Changing the root
    km.rootKeyStore.AddKey(privKey.ID(), "root", privKey)

    return privKey.ID(), nil
}
```

Listing 3: `keystoremanager.go:GenRootKey()`, key material in `privkey` not zeroed before garbage collection.

Exploit Scenario: A developer uses a cloud server shared with other virtual instances to publish application updates using Notary. An attacker discovers an exploit in the underlying virtualization software that allows the attacker to inspect the memory contents of all processes running on the host machine. When the developer uses Notary to publish an update, the developer's private key is decrypted and used to sign the update. The variable storing the private key is not zeroed after use, and the raw key material remains in memory after the signature operation is performed. The attacker inspects the memory of the Notary process, extracts the developer's private key, and publishes a malicious application update.

Short Term Solution: Overwrite all sensitive data as soon as it is no longer needed for processing.

Long Term Solution: Consider using a single mutable data type for all secret data, such as `[]byte`. Add a utility function that overwrites the contents of the data type with zeros, and call this utility function on the variables storing sensitive data immediately after their contents are no longer needed.

6. HSM PIN supplied via command line

Class: Configuration **Severity:** Low **Difficulty:** Low

Finding ID: CS-DRNT15-6

Targets: The PIN for the Hardware Security Module (HSM) used by Notary Signer.

Description: The Notary Signer exposes a signing service by running an RPC server accessible over HTTPS. The signer can make use of HSM to perform the signing operations. The HSM PIN is passed via the command line, and, as such, is visible to other users on the system (by looking at other processes on the system). Moreover, the PIN remains in shell history logs.

It should be noted that according one of the comments in `cmd/notary-signed/main.go`, the PIN will be loaded from a config file. Ideally, the signer administrator should enter the PIN every time the same way a passphrase is entered (interactively), without the need to store the PIN in the config file or expose it via an environment variable.

Exploit Scenario: A local user on the host that is running the `notary-signer` service runs `ps aux` and can observe the commands run by root and their command-line arguments. They learn the HSM PIN and are able to authenticate to it to perform operations on the HSM directly. They use this access to exploit one of the documented insecurities[4] in PKCS#11 interfaces to extract keys stored in the HSM.

Short Term Solution: Instead of passing the PIN via command line, have the user enter the PIN via standard input or retrieve it from a configuration file. If a configuration file is used, warn against non-interactive password use in the system logs, and have Notary check on startup that its permissions allow least access.

Long Term Solution: Document the threat model, and if protection against local users is desired. If these concerns are considered out-of-scope, mark this and finding 7 on the next page as accepted risks.

Note: Docker indicates that no trust is placed in the HSM PIN, and as such, its disclosure does not affect the security of the system.

[4] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.3442

7. Named Collections Directory is World Readable

Class: Data Exposure **Severity:** Informational

Finding ID: CS-DRNT15-7

Targets: Notary Client

Description: The Notary Client uses the `~/.docker` directory to store the current state of the Notary client. For example, if a `example.com/scripts` named collection is initialized, a directory `~/.docker/trust/trusted_certificates/example.com` is created. As `~/.docker/trust/trusted_certificates` and its sub-directories are world readable, another user on the system can peek into which named collections were initialized by the original user. It may be desirable to keep these names private to avoid unnecessary information leakage.

Solution: Use the existing `private` FileMode when creating directories under `~/.docker`, rather than `visible`.

8. Failure to Load Notary Server TLS Certificate Disables TLS

Class: Error Reporting **Severity:** Informational

Finding ID: CS-DRNT15-9

Targets: `notary/server/server.go:Run()`

Description: The server's TLS certificate and key pair are loaded in the `Run()` function of `server.go`, but if either fails to load for any reason, the server will fail open and will serve plaintext HTTP without TLS. This may mislead users into believing the service is operating normally when that is not the case.

This situation could be triggered in any number of normal situations, such as permissions errors on the private keyfile (such as attempting to run the daemon under a different user than the owner of the key file) or typos in the path to the certificate or key file.

Solution: Fall back to HTTP without TLS only when explicit an HTTP configuration indicates so; otherwise raise an error and abort start-up.

Note: This issue was resolved after the completion of the engagement.

Appendices

A Non-Security Suggestions

During the review, CS documented several suggestions that do not directly pose any security concern, but may be useful in a general-purpose Update Framework library.

A.1 Defense in Depth

A few areas were noted where the protocol specification or the implementation could preform some defense-in-depth checks.

1. The `ValidTUFType` function could check that the type is the type expected as well as being some valid type.
2. If a client knows about multiple hash algorithms, there is no reason not to validate the documents against the combination of all of them, rather than choosing the first the client knows about. This technique can provide resilience when a hash algorithm weakens.

A.2 Unusual Self-DoS

If `gotuf` were used as a general-purpose library, a user could inadvertently lock themselves out of the system in creative ways. While not strictly necessary, the library could check for and protect against these edge cases.

1. If a root rollover occurs, and the new root contains two keys (not one) the "as long as one succeeded" behavior of `signed.VerifyRoot` allows the second key to get stored in the database without a self-signature. This function could be changed to accept a 'threshold' variable, which on the first invocation would be '1' and on the second invocation would be the length of `allValidCerts`.
2. It does not appear that there is any protection against someone foot-gunning themselves and signing a document with a threshold for more keys than are specified. This would prevent any future documents from being accepted, as the threshold can never be met.

B Key Compromise Analysis

To determine the impact of certain vulnerabilities and their difficulty of exploitation, CS detailed the security mechanisms used to protect each key, as well as the impact of different key compromises. The first list, below, details how each authority key is used and protected. The following table, on the next page, outlines several scenarios of key compromises, different attacker capabilities to perform a man-in-the-middle attack, and whether each scenario allows different types of attacks.

- Timestamp Key - this key is administered by Docker. It is kept in a backend service that is designed to be modular, such that it may be protected by a HSM, and so that compromise of any front-end webservers do not lead to compromise of the key. The key is used when a user is authenticated to Docker, thus compromise of a user's account, or any vulnerability that would allow user impersonation may result in the Timestamp key being exercised in an unauthorized manner.
- Snapshot and Target Keys - these keys held and administered by the user, and thus compromise of them is largely dependent on the end-user security measured taken by the user. They are password protected, and the Notary applications do not have any options that enable exporting them in a plaintext format.
- Root Key - this key is also administered by the user, but Docker strongly recommends that this key be kept offline.

Two scenarios are presented, one where an attacker compromises a key and maintains persistent network access to a victim, and one where an attacker compromises a key and has only temporary network access to a victim.

	Malicious Updates	Freeze Attack	MetaData Inconsistency Attack	Denial of Service
Timestamp Key	No	Yes, limited by earliest expiry	No	Yes, finding 3
Snapshot Key	No	No	No	↓
Timestamp & Snapshot	No	Yes, limited by earliest target expiry	Yes	↓
Targets Key	No	No	No	↓
Snapshot & Target	Likely[5]	Yes, limited by root expiry[12]	Yes, limited by root expiry[12]	↓
Timestamp, Snapshot, & Target	Yes, temporary	Yes, limited by root expiry[12]	Yes, limited by root expiry[12]	↓
Root Key	Yes, permanent[34]	Yes	Yes	Yes[3]

Table 1: Temporary Network Access

	Malicious Updates	Freeze Attack	MetaData Inconsistency Attack	Denial of Service
Timestamp Key	No	Yes, limited by earliest expiry	No	Yes, finding 3
Snapshot Key	No	No	No	↓
Timestamp & Snapshot	No	Yes, limited by earliest target expiry	Yes	↓
Targets Key	No	No	No	↓
Snapshot & Target	Likely[5]	Yes, limited by root expiry[12]	Yes, limited by root expiry[12]	↓
Timestamp, Snapshot, & Target	Yes, limited by root expiry[2]	Yes, limited by root expiry[2]	Yes, limited by root expiry[2]	↓
Root Key	Yes	Yes	Yes	Yes[3]

Table 2: Persistent Network Access

[1]: Because no maximum expiry values specified at the Root level, a far-future document will be cached and used indefinitely.
[2]: Finding 2 details how, prior to patching by Docker, a Root update would not occur even if the Root document was expired.
[3]: An attacker could purposefully roll the root key to a new one.
[4]: Note that although indicated as "permanent", an attacker would also need to have persistent access to placing files into the *software repository* to actually exploit this.
[5]: Compromise of only the Snapshot & Target keys does not allow malicious updates, as the attacker must also compromise the Timestamp key. However, Docker operates the Timestamp key on behalf of a user, and an attacker who compromises the user's keys would likely be able to compromise the user's Docker account as well, and fool Docker into operating that key to achieve malicious updates.