Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report Nahoft Encryption Software 11.2020

Cure53, Dr.-Ing. M. Heiderich, Dr. N. Kobeissi

Introduction

TBD

Scope

- **Code Audits & Cryptography Reviews against Nahoft File Encryption Application**
  - In scope is the Nahoft Offline File Encryption Application written Java, Kotlin
  - Cure53 will be given Binaries, access to source Code, working builds, necessary Documentation and other means of access to complete the test with the coverage expected by U41
  - Cure53 will further be given information about the expected threat model and security promises to be able to model a realistic attacker scenario and audit the software accordingly
- **Sources were shared with Cure53**
- **Binaries were shared with Cure53 (APK)**
- **Test-supporting Material was shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *U4I-03-001*) for the purpose of facilitating any future follow-up correspondence.

U4I-03-002 Crypto: Steganography Techniques Extremely Ineffective (*Critical*)

The Nahoft application is described as *"intended for users who are in a country which has experienced a total Internet shutdown and the primary means of communication is through adversary-controlled messaging applications. The goals of the application are to encrypt the messages so that the adversary cannot read them and obfuscate the encrypted messages so that the adversary will not block them as they travel through the messaging application."*

During discussions with Cure53, U4I provided conflicting statements regarding the obfuscation requirements placed on Nahoft: at the outset of the audit, U4I indicated the following:

"The main threat will be that the government tries to read encrypted messages. To do this, they must first know that there is a hidden message there. ***So that's why we thought that maybe there is a way to hide the message in such a way that it was not even possible to detect the existence of an encrypted message.*** *[...]* ***Because if the government finds out that there is a hidden message here and can't read the messages, it will go to the app users.****"*

During the audit, U4I expressed the following, which contradicts its initial statement on steganographic guarantees:

"As far as the obfuscation, ***more advanced obfuscation is possible, but this level of obfuscation was considered by U4I to be sufficient for the project.****"*

Nevertheless, during the audit, it was found that the level of "obfuscation" (more accurately described as "steganography"[1]) poses critical issues for the Nahoft application's use case and could potentially put Nahoft users in immediate danger with little effort or preparedness on behalf of the stated adversary.

[1] https://www.merriam-webster.com/dictionary/steganography

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Let us suppose that Nahoft is given a small, roughly 200-word plaintext, along with the following photo of a kitten in which it is asked to “hide” the produced ciphertext:

Fig.: A picture of a kitten, without a hidden message involved

Nahoft will produce the following output, which the user is instructed to share:

Fig.: A picture of a kitten, including the hidden message

Effectively, the ciphertext is “hidden” not *within* the photo, but as a clearly encoded border *surrounding* the provided photo. As far as is known, there exists no application that produces an image border resembling the one produced by Nahoft. This renders Nahoft-produced images immediately identifiable. This is severely problematic for a number of reasons:

- **Nahoft-shared ciphertext is trivial to detect algorithmically.** A simple algorithm could scrape social media for Nahoft-produced photos, and immediately flag posts and users for future investigation.
- **Nahoft-shared ciphertext immediately identifies Nahoft usage.** Anyone sharing a Nahoft-produced photo is almost certainly attempting to communicate in a manner of interest to Iranian intelligence authorities (which the Nahoft application explicitly labels as the adversary).

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Using Nahoft effectively paints a target on the back of any Nahoft user in a way that can be automated and which is extremely easy to identify. Given that the Nahoft application ships with Persian-language manuals advising its usage to combat authoritarian government institutions in the Islamic Republic of Iran, it is, in its current state, an extremely dangerous tool that must not be deployed under any circumstances before more effective steganography techniques are implemented.

The text-based version of Nahoft "steganography" is also trivial to profile using the techniques mentioned above, due to the fact that it pads output to a multiple of 256 bytes and due to the identifiable nature of non-semantic, non-syntactic pseudorandom Persian alphabet usage.

It is strongly recommended to not deploy Nahoft under any circumstances before effective steganography, especially one that is resistant to automated, algorithmic detection, is employed. Research on JPEG-compression-resistant steganography is widely available:

- *PM1 Steganography in JPEG Images Using Genetic Algorithm* (by Lifang Yu, Yao Zhao, Rongrong Ni and Zhenfeng Zhu)
- *JPEG Compression Immune Steganography Using Wavelet Transform* (by J. Xu, A.H. Sung, P. Shi and Q. Liu)
- *JPEG Compression Steganography and Cryptography Using Image-Adaptation Technique* (by Meenu Kumari, A. Khare and Pallavi Khare)

In addition to the above research articles, free-access tutorials are available demonstrating the feasibility of JPEG-compression-resistant steganography.[2]

U4I-03-003 Crypto: Android Keystore Employed Partially *(Low)*

It was found that the Android Keystore system was only partially employed throughout the Nahoft stack. While short-term keys were stored in the Nahoft keystore, key pairs (specified in codex/Encryption.kt) were not. The developers gave the following rationale for not including long-term key pairs inside the Android Keystore:

"Note: The AndroidKeystore does not support ECDH key agreement between EC keys. The secure enclave does not appear to support EC keys at all, at this time. Therefore, we store keys in the EncryptedSharedPreferences instead of the KeyStore. This can be revised when the AndroidKeystore supports the required functionality."

[2] https://security.stackexchange.com/questions/224201/what-steganographic-tech...ive-lossy-compressio

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

However, as of Android API version 23 (which Nahoft adopts as its compilation target), EC key generation and Diffie-Hellman operations appear to be supported.[3]

As a minimum, it is recommended that EC keys be stored within the Android Keystore, with additional effort undertaken towards migrating Diffie-Hellman operations as well. While the Keystore currently supports only certain named curves (P-224 (*secp224r1*), P-256 (aka *secp256r1* and *prime256v1*), P-384 (aka *secp384r1*), P-521 (aka *secp521r1*)), migrating towards one of these curves could prove beneficial given Nahoft's mobile (and Android) centric use case.

U4I-03-005 Crypto: Fake Passcode Detectable by Adversary *(Medium)*

Nahoft offers a feature that allows users *"to set a passcode required to access the app, as well as a secondary passcode that, when entered, will delete all of the information such as messages and keys."*

While this secondary passcode functionality is judged effective in deleting all user data, its usage is likely to be immediately detectable by an adversary, since it is overwhelmingly unlikely that a user will hand over a Nahoft instance with absolutely no data within the application. The adversary may then be led to reasonably suspect that a secondary passcode was employed.

It is recommended to, instead of deleting all data, provide one of the following alternative functionalities:

- Replace all data with "dummy data" that is pre-defined by the user in advance.
- Delete a majority of the data while leaving a small number of entries untouched, chosen at random.

Of the above two options, the former is likely to produce the best results: it does not reveal the usage of the secondary passcode to the adversary while also not revealing sensitive information to the adversary.

U4I-03-006 Crypto: No Passcode Strength Enforced *(Medium)*

It was observed that the Nahoft application did not enforce any notion of passcode strength: simple passcodes such as *0*, *000*, *passcode*, and *password* were all accepted by the application. This could lead to scenarios in which users choose easily guessable passcodes without being aware of the risk.

[3] https://developer.android.com/training/articles/keystore

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected File:
Nahoft/activities/PasscodeActivity.kt

Affected Code:
```
private fun updateInputs(passcodeRequired: Boolean) {
	[...]
	// code enforcing passcode strength should be added here
}
```

It is recommended to employ a password strength measurement library, such as zxcvbn, in order to better determine legitimate input passcodes.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

U4I-03-001 Crypto: Encryption Functions Fails Silently *(Low)*

It was found that the core encryption and decryption functions, called by much of the Nahoft stack, fail silently in the event of an irregular nonce.

Affected File:
app/src/main/java/org/nahoft/codex/Encryption.kt

Affected Code:
```
fun encrypt(encodedPublicKey: ByteArray, plaintext: String): ByteArray? {
	val plaintTextBytes = plaintext.encodeToByteArray()
	val nonce = Random().randomBytes(SodiumConstants.NONCE_BYTES)
	val friendPublicKey = PublicKey(encodedPublicKey)
	val privateKey = ensureKeysExist().privateKey
	val result = SodiumWrapper().encrypt(
		plaintTextBytes,
		nonce,
		friendPublicKey.toBytes(),
		privateKey.toBytes())
	return if (result.size <= nonce.size) {
		null
	} else {
		result
	}
}
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
fun decrypt(friendPublicKey: PublicKey, ciphertext: ByteArray): String? {
	val keypair = ensureKeysExist()
	val result = SodiumWrapper().decrypt(ciphertext,
	friendPublicKey.toBytes(), keypair.privateKey.toBytes())
	return if (result.size <= SodiumConstants.NONCE_BYTES) {
		null
	} else {
		String(result)
	}
}
```

The null value is then manually handled at the *MessageActivity* layer:

```
private fun loadMessageContent() {
	message_sender_text_view.text = getString(R.string.sender_label,
	message.sender?.name)
	val senderKeyBytes = message.sender?.publicKeyEncoded
	if (senderKeyBytes != null) {
		val senderKey = PublicKey(senderKeyBytes)
		val plaintext = Encryption(this).decrypt(senderKey,
		message.cipherText)
		if (plaintext != null) {
		message_body_text_view.text = plaintext
		} else {
			applicationContext.showAlert(getString(R.string.alert_text_u
			nable_to_decrypt_message))
			deleteMessage(this, message)
			finish()
		}
	} else {
		print("Failed to get sender public key for a message")
		return
	}
}
```

It is recommended that decryption failure explicitly throws an error at the level of the decryption functionality, instead of custom decryption failure notifications needing to be implemented at the user interface layer every time that the encryption or decryption functions are called.

U4I-03-004 Crypto: Data “Clearing” Not Ensured in Memory/Storage *(Info)*

It was observed that the Nahoft application “cleared” sensitive user-data by “zero-filling” it both in memory and on disk. While this technique is effective at rendering data immediately inaccessible, it should be noted that it is not guaranteed to immediately erase data from device RAM as well as from disk cache, garbage collection, backups, etc.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected File:
Nahoft/Persist.kt

Affected Code:

```
fun clearAllData() {
        if (friendsFile.exists()) { friendsFile.delete() }
        if (messagesFile.exists()) { messagesFile.delete() }
        friendList.clear()
        messageList.clear()
        // Overwrite the keys to EncryptedSharedPreferences
        val keyHex =
        "0000000000000000000000000000000000000000000000000000000000000000"
        encryptedSharedPreferences
        .edit()
        .putString("NahoftPrivateKey", keyHex)
        .putString(publicKeyPreferencesKey, keyHex)
        .apply()
        // Remove Everything from EncryptedSharedPreferences
        encryptedSharedPreferences
        .edit()
        .clear()
        .apply()
        status = LoginStatus.NotRequired
}
```

No recommendation is suggested as a matter of course for this issue, but Nahoft developers (and users) should be made aware of the limitations of the approach employed by Nahoft to deal with key and local materials erasure.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Cryptography Review

The Nahoft application relies on a combination of cryptography and steganography in order to achieve its goal: allowing users to create and communicate short encrypted messages that are hidden within innocuous media such that an adversary cannot detect the usage of encryption or the presence of encrypted messages.

In this section, we summarize the findings of this report both in terms of Nahoft's cryptography and steganography implementations, and conclude with considerations regarding the viability of the current Nahoft application given its stated use case scenarios.

Review of Cryptographic Primitives and Protocol

The Nahoft application employs the well-specified and well-audited Libsodium cryptographic library, providing it in a simple surrounding Kotlin wrapper. The intent is to use Libsodium's cryptographic constructions in order to construct a "naïve Diffie-Hellman protocol" where parties perform basic key agreement over elliptic-curve Diffie-Hellman. This shared secret is used directly in order to encrypt data with Libsodium's authenticated encryption constructions.

While no issues could be spotted with the cryptographic protocol targeted by the Nahoft application, minor issues were detected with regards to how Nahoft stores long-term keys on device without using the Android Keystore system (U4I-03-003), the lack of proper error handling on certain cryptographic functionality (U4I-03-001) and unclear expectations on the application's "data wiping" functionality (U4I-03-004). All of these issues are simple to address, and no severe findings, or findings with an unclear resolution path, were found in Nahoft's cryptography stack.

Data Obfuscation and Steganography Considerations

While Nahoft's cryptography stack exhibited no serious issues, it is in Nahoft's steganography stack that serious shortcomings were detected, to the degree of seriously harming the application's ability to meet its stated use case scenarios and not potentially harm the safety of its users.

As stated in U4I-03-002:

> *"The Nahoft application is described as "intended for users who are in a country which has experienced a total Internet shutdown and the primary means of communication is through adversary-controlled messaging applications. The goals of the application are to encrypt the messages so that the adversary*

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

> *cannot read them and obfuscate the encrypted messages so that the adversary will not block them as they travel through the messaging application."*
>
> *During discussions with Cure53, U4I provided conflicting statements regarding the obfuscation requirements placed on Nahoft: at the outset of the audit, U4I indicated the following:*
>
> *"The main threat will be that the government tries to read encrypted messages. To do this, they must first know that there is a hidden message there.* ***So that's why we thought that maybe there is a way to hide the message in such a way that it was not even possible to detect the existence of an encrypted message.*** *[...]* ***Because if the government finds out that there is a hidden message here and can't read the messages, it will go to the app users.****"*
>
> *During the audit, U4I expressed the following, which contradicts its initial statement on steganographic guarantees:*
>
> *"As far as the obfuscation,* ***more advanced obfuscation is possible, but this level of obfuscation was considered by U4I to be sufficient for the project.****"*
>
> *Nevertheless, during the audit, it was found that the level of "obfuscation" (more accurately described as "steganography"[4]) poses critical issues for the Nahoft application's use case and could potentially put Nahoft users in immediate danger with little effort or preparedness on behalf of the stated adversary."*

Issue U4I-03-002 then proceeds to provide examples demonstrating how Nahoft's steganography is not only immediately visible, but is such that it can be easily targeted by automated algorithms that would be able to detect images containing Nahoft-encrypted ciphertext with a high degree of accuracy.

As the issue discusses, Nahoft's current "obfuscation" techniques do not meet the bar for steganography. The issue concludes by providing a series of research and practical techniques that show how actual steganography may be accomplished in ways that survive JPEG image compression, which is commonly present on Internet and social media platforms.

Furthermore, issues U4I-03-005 and U4I-03-006 discuss shortcomings in Nahoft's passcode system, which could see users choosing passcodes of insufficient strength, or inadvertently revealing the usage of a "fake" passcode to the adversary.

[4] https://www.merriam-webster.com/dictionary/steganography

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Viability of the Current Nahoft Application Use Case Scenarios

It is critical to put Nahoft's current functionality in the context of its stated adversary: the Islamic Republic of Iran's Ministry of Intelligence (MOIS). The MOIS is particularly well known for carrying out assassinations, incarceration and other quelling strategies when targeting political dissidents within Iran.[5]

This constitutes an exceptionally difficult, aggressive and well-equipped adversary. Given the nature of the application's stated adversary, it is worth strongly considering whether steganography in the form of trivially profilable image borders is appropriate. It is overwhelmingly likely that the Nahoft application, in its current state, would produce image artifacts that would quickly reveal Nahoft usage to observers and adversaries, which would almost certainly put Nahoft users in serious danger.

Until these issues are addressed, publishing the Nahoft application for public use is strongly discouraged, and may constitute extreme harm to its users if they are encouraged to use it in a politically sensitive context within the Islamic Republic of Iran.

Conclusions

TBD

Cure53 would like to thank Milad Keshtan and Reza Ghazinouri as well as the rest of the U4I team as well as Adelita Schule and Brandon Wiley of Operator Foundation for their good project coordination, support and assistance, both before and during this assignment.

[5] https://www.tandfonline.com/doi/abs/10.1080/08850609708435351