Ricochet Security Assessment

Ricochet

February 15, 2016 – Version 1.2

Prepared for

John Brooks

Prepared by

Jesse Hertz

Patricio Jara-Ettinger

Mark Manning

©2016 – NCC Group

Prepared by NCC Group Security Services, Inc. for Ricochet. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Executive Summary

Synopsis

Ricochet and the Open Technology Fund engaged NCC Group to perform an assessment of the Ricochet anonymous messaging application. Two NCC Group consultants performed the assessment between November 16th and 25th. The engagement was conducted as a source code review with a focus on identifying common C++ vulnerabilities, privacy influencing vulnerabilities, as well as to provide recommendations based on security gaps vs best practices.

Ricochet expressed particular interest in the discovery of vulnerabilities that, if exploited, could lead to the de-anonymization or exploitation of Ricochet users. NCC Group focused its testing efforts on attempting to find vulnerabilities with these impacts.

Scope

The testing team reviewed the provided C++ source code as well as the design documents. The QML files that are used for UI layout were deemed out of scope. Ricochet relies upon Tor and Tor hidden services but Tor itself was not included in scope. Cursory dynamic analysis of Ricochet was performed but additional testing is recommended. AFL[1] was used to fuzz the application, and several simple utilities for fuzzing Ricochet with AFL are provided in Appendix B on page 20:

- A simple client written in Python used to feed the AFL-generated inputs into the application.
- A series of small modifications to Ricochet to facilitate fuzzing via AFL.
- A small C shim to launch the Python client, and then `execve()` to Ricochet.

Key Findings

The assessment identified multiple areas of improvement that include one issue given a High risk rating. Many of the findings are provided as a defense-in-depth approach for developers to continue to focus their effort. The source code assessment identified input validation issues that were already known to the developer, and have issues filed in GitHub. These include:

- Issues with HTML being included in contact requests, leading to de-anonymization if the request is accepted.
- Issues with Unicode processing leading to homograph or other phishing attacks, both on links or in contact nicknames. These could allow de-anonymization or compromise of user privacy.

[1] http://lcamtuf.coredump.cx/afl/

Project Dashboard

nccgroup

Target Metadata

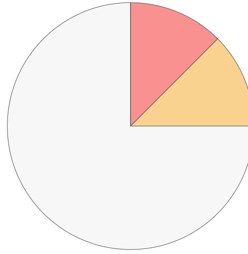
Name	Ricochet Security Audit
Type	Anonymous Instant Messenger
Platforms	C++, QT, Protobuf

Engagement Data

Type	Source Code Review
Method	White Box
Dates	2015-11-16 to 2015-11-22
Consultants	2
Level of effort	16 person-days

Vulnerability Breakdown

Critical Risk issues	0
High Risk issues	1
Medium Risk issues	1
Low Risk issues	0
Informational issues	6
Total issues	**8**



Category Breakdown

Cryptography	3
Data Exposure	2
Data Validation	3

Key

Critical High Medium Low Informational

Best Practices Evaluation

Ricochet's security posture was evaluated in several key areas. Overall, Ricochet was found to be a strong application that met best practices for a privacy-focused C++ program. However, several areas were identified in which improvements could greatly enhance the security of Ricochet.

Rating Explanation

- **Satisfactory**: Meets or exceeds industry best practice
- **Fair**: May not be in total compliance with best practices, but no directly actionable issues were identified
- **Needs improvement**: Fails to comply with best practices, and contains actionable flaws in this area

Input Validation

As designed:	As implemented:
Fair	**Needs improvement**

Best Practice: A famous dictum in security is "anything that takes user input can be hacked". While a bit hyperbolic, processing input from users is the most common area in which security vulnerabilities are found.

Sources of user input should be identified, and validated according to the context of the input. Various strategies for validation include: logic checks, sanitization, whitelisting, and output-encoding. Which strategies are appropriate depends on the context of the input.

Evaluation: Ricochet's input validation varies greatly depending on what layer is being examined:

- At the protocol layer, there are a number of checks to ensure that incoming messages are properly sized, and are valid according to the protocol state machine. These include strong defensive programming techniques, such as ensuring that a connection's channel is the expected value.
- At the application layer, input validation is lacking. Vulnerabilities such as finding NCC-Ricochet Security Audit-007 on page 10 or finding NCC-Ricochet Security Audit-006 on page 11 may compromise the privacy of a user, due to insufficient validation of the contents of messages.

Recommendation:

- At the protocol layer, continue using defensive programming techniques, such as checking that any protocol-level state changes are allowed, incoming requests are well-formed, incoming messages are on the correct channel, incoming requests on a channel are on the correct connection, and so forth. Particular care should be taken whenever a change to the underlying connection object for a channel is made (this is done very rarely), as a mistake here could lead to messages going to a different party.
- At the application layer, apply strict validation to all user input. Again, defensive programming can be used here, by combining a whitelist approach of allowable input with context-appropriate encoding of potentially dangerous characters before they are used. For more information, see finding NCC-Ricochet Security Audit-007 on page 10 and finding NCC-Ricochet Security Audit-006 on page 11.

Cryptography

As designed:	As implemented:
Needs improvement	**Fair**

Best Practice: Cryptography is one of the easiest elements to get wrong in security. Very small mistakes in an implementation can lead to exploitable flaws that allow compromise of authenticity, anonymity or confidentiality.

Because of this accepted difficulty, it is highly recommended to use standard implementations of cryptographic techniques, especially those that have been extensively audited by the cryptography community. In addition, protocol design should aim to eliminate cryptographic complexity, as it is easy to combine what are otherwise secure primitives in an insecure way.

Because Ricochet is meant to be a messaging service for those who need anonymity, its threat model should include state-level adversaries, who may have cryptanalytic techniques beyond even what is publicly known. As such, it is highly recommended to use the most up-to-date encryption schemes, and layer encryption when that is not possible. In addition, state-level adversaries may have the ability to obtain physical access to the machine, so care should be taken to protect data at rest using cryptography.

Evaluation: Ricochet provides transport layer security using Tor's hidden services.[2] While these do provide cryptographic protection (including perfect forward secrecy), several of the cryptographic primitives they are built on are no longer considered strong. Ricochet uses a protocol for authenticating the client to the server that introduces unnecessary cryptographic complexity. In addition, no encryption is present on data at rest.

Recommendation: Ricochet should take several steps to improve its cryptographic security:

- Rework the host-verification, as detailed in finding NCC-Ricochet Security Audit-001 on page 12 and finding NCC-Ricochet Security Audit-003 on page 14.
- Add application-layer cryptography, as detailed in finding NCC-Ricochet Security Audit-004 on page 15.
- Encrypt sensitive data at rest, as detailed in finding NCC-Ricochet Security Audit-005 on page 17.

In addition, it would be beneficial for Ricochet to implement Channel Binding with the hidden service link-layer encryption. By binding Ricochet's key exchange to the underlying transport, one can be assured that both parties of the connection are communicating on the same encrypted connection and further mitigate forwarding or man-in-the-middle attacks. In order to do this, Tor would need to add support for channel binding, similar to the tls-unique channel binding available in TLS.

Hardening/Sandboxing

As designed:	As implemented:
Needs improvement	**Fair**

Best Practice: In order to prevent vulnerabilities in an application from being exploited, various mitigation strategies can be taken. These include hardening the application (to make exploitation of vulnerabilities more difficult) and sandboxing the application (to limit the impact of successful exploitation).

Hardening strategies include the use of address sanitizer (which will cause the application to terminate when memory flaws are detected, although this introduces significant performance overhead), as well as using compiler-level hardening flags.

Sandboxing strategies include running the application inside a container/sandbox (such as LxC[3]/Docker[4] for Linux, or AppContainer[5]/Sandboxie[6] for Windows), or operating system access control profiles (such as AppArmor[7] for Linux or Seatbelt for OSX/iOS) to restrict what resources the application has access to and what operations it can perform.

[2] https://www.torproject.org/docs/hidden-services.html.en
[3] https://linuxcontainers.org/
[4] https://www.docker.com/
[5] https://github.com/appc/spec
[6] http://www.sandboxie.com/
[7] http://wiki.apparmor.net/index.php/Main_Page

Protecting the running Tor instance is paramount to the anonymity of Ricochet users. Sandboxing strategies could be employed to further protect Tor, its services, and specifically the control port which, if exploited, offers adversaries the opportunity to completely compromise the anonymity of the Ricochet user. This could be done by injecting a new set of malicious Directory Authorities or building a path to an adversary-owned Rendezvous Point for the hidden services to perform traffic analysis on. The aforementioned hardening tools can be applied to the protect the control port such as blocking all TCP connections to the control port via an AppArmor/ Seatbelt profile and changing the control port listener to run on a Unix socket.

Evaluation: Ricochet currently has experimental support for sandboxing, and ships with some profiles for doing so, although they are not enabled by default. There is discussion about sandboxing in issue #222.

Ricochet normally builds with AddressSanitizer, and uses various compiler hardening flags.

Recommendation: Continue looking into integrating Ricochet with various sandboxing solutions. Note that this may require a decent amount of customization to make it useable for non-technical users.

Consider additional mitigations to defend against a compromise via the tor control port including AppArmor/Seatbelt profiles that restrict TCP connections and using Unix sockets instead of a TCP listener.

Continue using address sanitizer in all builds. Continue using build flags that enable further hardening, see this guide for more information.

Memory Management

	As designed:	As implemented:
	Satisfactory	**Satisfactory**

Best Practice: Due to the memory-unsafe nature of C++, great care needs to be taken to avoid memory corruption issues. Two primitives are often used to violate memory-safety:

- Using "raw" memory structures, such as arrays or regions of memory obtained from malloc().
- Allowing an attacker to influence the position (or length) of access to these structures.

Instead of utilizing arrays/malloc, programmers should use wrappers around these that provide memory-safe semantics. When using memory-unsafe primitives directly, great care should be taken to validate the size of the allocation, as well as the size of the accesses.

Since C++ has manual memory management, objects are allocated and freed by hand, and are not reference counted. As such, it is very common when passing pointers around to have vulnerabilities such as doubles frees or use-after-frees.

In order to handle pointers in a secure way, updates to C++ introduced various kinds of smart pointers, which wrap raw pointers and create safer ways to share and manage pointers.

Evaluation: Ricochet does an excellent job of preventing errors by:

- Making extensive use of the `QByteArray` primitive, which allows array-like semantics in a memory-safe fashion (bound checking occurs automatically, and resizing occurs when necessary).
- Similarly, string operations are done using a mix of `QByteArray` and `std::string` objects, providing memory safety.
- When it is necessary to pass a buffer pointer and corresponding length to a function that operates on memory directly, the buffer and the length are received using the `.data()` and `.size()`/`.length()` functions, without performing additional arithmetic. This eliminates a wide variety of arithmetic errors that can lead to memory corruption.
- When sizes are read from user provided input (such as when reading the number of versions during version negotiation, or reading the size of the incoming message), the size is read into a small integer (either a `quint8` or a `quint16`) and then all calculations are done with word-size variables. This drastically lowers the likelihood of an arithmetic overflow/underflow happening in a subsequent calculation using these sizes (as they are orders of

magnitude below the word size on even a 32-bit architecture).

- In addition, all sizes read from user input are subsequently validated to be within sane bounds (for instance, the maximum size for a message is `UINT_MAX`), which greatly reduces the possibility of integer overflow/underflow bugs.
- Sizes are also validated to match the logic of the application (for instance, the message size is checked to be equal to the size returned by the call to `read()`).
- Rather than write code to parse messages as they come in off the wire (which is very frequently an area where memory corruption issues are encountered), Ricochet uses Google's protobuf library. This allows the programmer to specify the structure of messages in metadata files, and have the C++ code to parse/create messages automatically generated.
- Ricochet makes extensive use of Qt's smart pointer templates, such as `QSharedPointer`, `QScopedPointer`, `QExplicitlySharedDataPointer`, etc. Ownership of pointers is made explicit (both in the code and in the comments), allowing sharing of pointers in a straightforward and secure fashion.

Recommendation: Continue following best practices and defensive programming guidelines, as well as using existing solutions that provide memory-safe data-structures and utilities. Test ricochet (ideally using a fuzzer) while running it to detect some classes of memory errors.

Use of Unsafe C/C++ Functions

As designed:	As implemented:
Satisfactory	**Satisfactory**

Best Practice: Do not use C/C++ functions that are inherently unsafe, such as `gets()`, `strcpy()`, the » operator, etc. Do not use C/C++ functions that are easily misused, such as passing user-controllable input as the format-string to a `printf()` style function, or including user input in a call to `system()`.

MITRE defines two classes of weaknesses for this, CWE-242: Use of Inherently Dangerous Function, and CWE-676: Use of Potentially Dangerous Function.

Numerous guides have been written on this topic, and static analysis tools such as cppcheck or cpplint can be used to automatically check source code for these types of errors.

Evaluation: Ricochet does not use dangerous C/C++ functions or constructs.

Recommendation: C++ guidelines on use of dangerous functions should continue to be followed, and static analysis should be performed periodically on the codebase to ensure that dangerous code has not been introduced.

Table of Vulnerabilities

nccgroup

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and vulnerability categorization, see Appendix A on page 18.

Title	ID	Risk
Insufficient Validation in ContactRequest Allows De-Anonymization	007	High
Links May Contain Malicious Unicode Characters	006	Medium
Unnecessary Use of HMAC	001	Informational
Access To Local Socket Can Steal 32-Byte Files	002	Informational
Host Verification Weak Against State Level Adversaries	003	Informational
Lack of Application Layer Message Encryption	004	Informational
Unexploitable Buffer Overread in `CryptoKey::loadFromData`	008	Informational
Configuration/Metadata Files Stored on Disk Unencrypted	005	Informational

Vulnerability	**Insufficient Validation in ContactRequest Allows De-Anonymization**
Risk	High Impact: High, Exploitability: Medium
Identifier	NCC-Ricochet Security Audit-007
Category	Data Validation
Target	protocol/ContactRequestChannel.cpp

- `isAcceptableNickname`

Impact

A Ricochet contact request can contain a nickname/message with HTML sequences, or various obscure unicode characters. Trivially, this can be used to disclose the real IP address of a user if the user accepts the contact request. This could also be used to create a nickname designed to be visually similar to another nickname (a form of homograph attack). Alternatively, it may be possible to trigger operating-system level bugs when attempting to render the message.

Description

Contact requests can include both a short message, and a nickname for the contact. Some validation is performed of the nickname:

```
92    QVector<uint> chars = input.toUcs4();
93      foreach (uint value, chars) {
94          QChar c(value);
95          if (c.category() == QChar::Other_Format ||
96              c.category() == QChar::Other_Control ||
97              c.isNonCharacter())
98              return false;
99      }
```

Listing 1: Relevant source code from ContactRequestChannel.cpp

However, this does not rule out several other kinds of malicious nicknames:

- Nicknames containing HTML tags (which, as the UI is rendered using QML, will end up being rendered later on, leading to a potential disclosure of the user's real IP address). This is already a known issue but without the security context: See this pull request #274.
- A large portion of the unicode code points would not be caught by this filter, which could allow homograph attacks.

In addition, no validation (aside from a size check) is performed on the message sent in the contact request, opening up the user to the possibility of operating-system level bugs when trying to render strange unicode sequences (for instance: Apple's CoreText framework recently contained a vulnerability when rendering certain invalid unicode sequences).

This is a particular concern since anyone with knowledge of a user's Ricochet ID can send them a contact request.

Recommendation

Since anyone can send a contact request to a user if they are aware of the user's username, these fields should be treated with special care. Use a whitelist of allowable characters that is as small as possible and consider output encoding where applicable.

Vulnerability	**Links May Contain Malicious Unicode Characters**
Risk	**Medium** Impact: Medium, Exploitability: Medium
Identifier	NCC-Ricochet Security Audit-006
Category	Data Validation
Target	ui/LinkedText.cpp • `LinkedText::LinkedText`
Impact	A link can contain various obscure unicode characters. This could be used to create a link (to an attacker site) designed to be visually similar to a trusted site (a form of homograph attack). Visiting this link would cause the user's real IP address (and other metadata sent as part of a normal browser request) to be disclosed. Alternatively, it may be possible to trigger operating system level bugs when the link is copied to the clipboard.
Description	Ricochet automatically parses messages and when it detects a URL, it automatically formats it into a clickable link. However, no validation or output encoding is done for the body of the link (aside from a call to `.toHtmlEscaped()`, which only escapes HTML metacharacters). An attacker could send a link that appeared to be for a known safe website, but (using obscure unicode codepoints) actually pointed to an attacker-controlled site. Note that there is already a pull request open to address this. Alternatively, it may be possible to trigger an operating-system level bug when malicious data is copied to the clipboard (and such vulnerabilities have existed before within Microsoft Windows).
Recommendation	Restrict the set of characters that are allowed in the body of a link to the smallest possible set.. Consider using output-encoding on characters outside the normal printable range. Review the above pull request accept if it implements these recommendations.

Vulnerability	**Unnecessary Use of HMAC**
Risk	**Informational** Impact: High, Exploitability: Low
Identifier	NCC-Ricochet Security Audit-001
Category	Cryptography
Target	protocol/AuthHiddenServiceChannel.cpp: • `AuthHiddenServiceChannel::handleProof`
Impact	The cryptographic construction used to authenticate the client to the server is unnecessarily complex, which may open the protocol up to sophisticated cryptanalytic attacks by a state-level adversary.

Description

Tor hidden services are based off RSA keys. In order to authenticate the server (to the client), the server needs to prove that it is the hidden service specified by an 80-bit hostname. This hostname is the first half of the SHA1 hash of the hidden service's RSA public key. In order to prove that the server "owns" the hostname, the client sends a nonce, and the server returns the nonce signed with their private key, as well as a copy of their public key. The client can then check the public key verifies the signature on the nonce correctly, as well as hashes to the hostname specified. This construction is safe as long as an attacker cannot craft a public key whose SHA-1 hash collides (in the first half) with the legitimate server's, which is discussed further here: finding NCC-Ricochet Security Audit-003 on page 14.

Fundamentally, there is no reason that this same protocol cannot be used to verify the client as well; in Ricochet, both parties are identified by their hidden service hostname, and both parties exchange public keys. However, the way the server verifies the client involves a more complicated construction, where both the server and the client provide nonces and their public keys, and then an HMAC is computed. This HMAC is unnecessary (as it adds no additional protection), as well as possibly providing an additional foothold for a cryptanalytic attack (since now there are two nonces being used, as well as the possibility of an attack on HMAC-SHA256).

Recommendation

Copy the existing RSA-based protocol used to perform hidden service authentication. Since this is a core part of Tor hidden services, breaking this protocol would imply a compromise of Tor hidden services as a whole. Do not introduce additional cryptographic complexity unless it is needed.

Vulnerability: Access To Local Socket Can Steal 32-Byte Files

Risk **Informational** Impact: Low, Exploitability: Low

Identifier NCC-Ricochet Security Audit-002

Category Data Exposure

Target tor/TorControl.cpp:340

Impact A malicious process listening on what Ricochet believes is the Tor socket will be able to request any 32-byte file from Ricochet. This could be used for host-fingerprinting, or in extremely fortunate circumstances, compromising sensitive information.

Exploitation of this vulnerability would be very difficult, as it would require impersonating the Tor control socket.

Description As part of the protocol between Ricochet and the Tor binary, Ricochet authenticates with Tor over a local socket (referred to as the Tor control port). One option for this authentication is "cookie" based authentication, where a secret value is stored on the filesystem and then sent to Tor over the control port. The authentication message requesting a cookie contains a file path for the cookie, which is potentially dangerous, as it would allow disclosure of arbitrary files. This attack is mitigated by a size check to verify this file is the correct size as a cookie, however, this does not prevent disclosure of arbitrary 32-byte files.

```
340 /* Simple test to avoid a vulnerability where any process
341  * listening on what we think is the control port could trick
342  * us into sending the contents of an arbitrary file */
343 if (cookie.size() == 32)
344   data = auth->build(cookie);
345 else
346   cookieError = QStringLiteral("Unexpected file size");
```

Recommendation Store all cookie files in a specific directory (or a whitelist of allowed directories), and validate that the cookie file requested is in a valid directory.

A more comprehensive approach would be to eliminate using filenames and switch to an opaque reference (such as a hash); however, this would require re-architecting the Tor control protocol.

Vulnerability	**Host Verification Weak Against State Level Adversaries**
Risk	**Informational** Impact: High, Exploitability: Low
Identifier	NCC-Ricochet Security Audit-003
Category	Cryptography
Impact	An attacker who is able to create a keypair whose public key's SHA1 hash collides with the hostname of a targeted hidden-service, will be able to impersonate that hidden service. This is not an issue unique to Ricochet, but is an issue with Tor hidden services.

Description

Tor hidden services are authenticated using the first 80 bits of the SHA1 hash of the hidden service's public key. While there are no publicly known attacks against this, there are several reasons why this may be worrying:

- As Ricochet is an anonymous messaging application designed for privacy, its threat model needs to include state-level adversaries.
- SHA1 is no longer considered to be cryptographically strong, public research has demonstrated weaknesses in its collision resistance. It is likely that state-level adversaries have better attacks than the publicly known ones. In addition, the hostname is only the first half of the SHA1 hash, which further reduces the difficulty of finding a collision.
- The RSA keys used in Tor are 1024-bit keys, which are no longer believed to be secure against state-level adversaries.

Recommendation

Work with Tor to improve the security of hidden services (currently, Tor is working on overhauling hidden services to use stronger cryptography). In the meantime, consider implementing additional verification, such as having the Ricochet ID include the full SHA1 hash, or a SHA256 hash of the public key.

Vulnerability	**Lack of Application Layer Message Encryption**
Risk	**Informational** Impact: Medium, Exploitability: Low
Identifier	NCC-Ricochet Security Audit-004
Category	Cryptography
Impact	An attack on the Tor hidden services transport layer may allow compromise of messages, or message metadata.
Description	Messages passed over Ricochet are only protected by the encryption present in the Tor hidden services transport layer. As is described in the Ricochet protocol document, "Connections are encrypted end-to-end, using the server's key and a DHE handshake to provide forward secrecy." However, Tor hidden services have been subjected to various types of attacks with varying degrees of success. Whether or not these attacks were successful or relevant to Ricochet, it's commonly accepted that adversaries are investing substantial resources to attack hidden services. In order to maximize protection against state-level adversaries while waiting for the next generation of hidden/onion services protocol to address some of known short-comings, it may be beneficial to add application-layer encryption of messages as well. Note that there is already an open feature request proposing exactly that.
Recommendation	Consider layering strong application-layer encryption on top of the Tor hidden services transport-layer encryption. Review the aforementioned feature request to determine its viability.

Vulnerability **Unexploitable Buffer Overread in `CryptoKey::loadFromData`**

Risk **Informational** Impact: High, Exploitability: None

Identifier NCC-Ricochet Security Audit-008

Category Data Validation

Target utils/CryptoKey.cpp:75

Impact While this is currently unexploitable, if code is refactored so that public keys are presented in a different format (such that it hits the vulnerable path with a non-null-terminated string), a buffer overread would occur, likely leading to a crash.

Description In `CryptoKey::loadFromData()`, the following code is used to parse a PEM-formatted key:

```
75 if (format == PEM) {
76     BIO *b = BIO_new_mem_buf((void*)data.constData(), -1);
77 
78     if (type == PrivateKey)
79         key = PEM_read_bio_RSAPrivateKey(b, NULL, NULL, NULL);
80     else
81         key = PEM_read_bio_RSAPublicKey(b, NULL, NULL, NULL);
82 
83     BIO_free(b);
```

In the above code snippet, `BIO_new_mem_buf()` is called with a length parameter of -1, indicating that the length of the buffer should be calculated by calling `strlen()` on the input data. However, the input data in this case is from a `QByteArray`, which is not guaranteed to be null-terminated (for instance, when created using the `QByteArray::fromRawData()` method). If a non-null-terminated buffer was provided as the data argument, `strlen()` would continue reading past the end of the buffer, possibly leading to an application crash.

Currently, the only invocation of `loadFromData()` is in `AuthHiddenServiceChannel::handleProof()`, which is unexploitable for two reasons:

- The buffer is constructed using the default `QByteArray()` constructor (guaranteeing null termination), rather than `fromRawData()`.
- The invocation of `loadFromData()` sets the format parameter to `CryptoKey::DER`, meaning the vulnerable section is currently dead code.

Recommendation Since the data is being stored in a `QByteArray`, the size is already known, so `BIO_new_mem_buf()` could be called as:

```
BIO *b = BIO_new_mem_buf((void*)data.constData(), data.size());
```

However, as this code is currently unused, it could also just simply be removed.

Vulnerability	**Configuration/Metadata Files Stored on Disk Unencrypted**
Risk	**Informational** Impact: Low, Exploitability: Low
Identifier	NCC-Ricochet Security Audit-005
Category	Data Exposure
Impact	An attacker with access to the local filesystem may be able to read or modify various configuration or metadata files used by Ricochet. These settings files include information about known contacts, as well as the configuration of Ricochet and Tor. In a threat model that includes state-level adversaries, physical access to the device cannot be ruled out, and as such, the threat of an adversary seizing a laptop and attempting to read the drive contents should be taken seriously. It is worth noting that if the adversary is able to write to the filesystem, much simpler and more severe attacks are possible.
Description	Ricochet stores various pieces of information on the filesystem unencrypted, including its settings files (which contain known contacts), the cookie files used to authenticate to the Tor control port, and the .torrc file used to maintain the Tor instance configuration. While an attacker with write access to the filesystem could simply backdoor Ricochet (or the system itself), an attacker with read access (for instance, through a weakness in a different application that allows file disclosure) would be able to view these files.
Recommendation	When possible, encrypt sensitive files when storing them on the filesystem. In order to deal with the threat of a state-level adversary, these should be encrypted using strong encryption, with a key created from a password entered at runtime using a cryptographically strong key derivation function. As a short term mitigation, consider using the OS user-account crypto-API's to protect these files. NOTE: This does not defend against attacks where adversaries are able to seize the laptop but would protect the files without requiring a separate password to be entered on start. For some of these (for instance the .torrc file), this may not be possible without making changes to Tor.

Appendix A: Vulnerability Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing vulnerabilities. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a vulnerability poses to the target system or systems. It takes into account the impact of the vulnerability, the difficulty of exploitation, and any other relevant factors.

Critical Implies an immediate, easily accessible threat of total compromise.

High Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

Medium A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

Low Implies a relatively minor threat to the application.

Informational No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable vulnerability.

Impact

Impact reflects the effects that successful exploitation upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational loesses.

High Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

Medium Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

Low Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a vulnerability. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High Attackers can unilaterally exploit the vulnerability without special permissions or significant roadblocks.

Medium Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the vulnerability.

Low Exploitation requires implausible social engineering, a difficult race condition, guessing difficult to guess data, or is otherwise unlikely.

Category

NCC Group groups vulnerabilities based on the security area to which those vulnerabilities belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

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 race conditions, locking, or order of operations.

Appendix B: Fuzzing

AFL[8] was used for fuzzing of the application, and a simple Python client was used to feed the generated inputs to Ricochet.

In order to allow AFL to keep track of unique code paths, Ricochet needs to be compiled using AFL. To ease the fuzzer development, the following snippets were added to the following files:

Connection.cpp was edited to exit the application gracefully after the fuzzed input was submitted:

```
void ConnectionPrivate::socketReadable()
{
    char quit[] = "exitgracefully";
    char done[21];
    socket->peek(done, 21);
    if (strcmp(quit, done) == 0)
    {
        exit(0);
    }
    if (!handshakeDone) {
    ...
```

AuthHiddenServiceChannel.cpp was edited to accept invalid signatures:

```
void AuthHiddenServiceChannel::handleProof(const Data::AuthHiddenService::Proof &message)
    ...
    if (!ok) {
        qWarning() << "Signature verification failed on" << type();
        result->set_accepted(true);
    } else {
        result->set_accepted(true);
        qDebug() << type() << "accepted inbound authentication for" << publicKey.torServiceID();
    }
    ...
```

UserIdentity.cpp was edited to have Ricochet always listen on a predictable port:

```
UserIdentity::UserIdentity(int id, QObject *parent)
    ...
    // Generally, these are not used, and we bind to localhost and port 0
    // for an automatic (and portable) selection.
    QHostAddress address(m_settings->read("localListenAddress").toString());
    if (address.isNull())
        address = QHostAddress::LocalHost;
    quint16 port = (quint16)m_settings->read("localListenPort").toInt();
    port = 55555;
    ...
```

[8] http://lcamtuf.coredump.cx/afl/

The Python client takes the content of a file as input (generated by AFL) and places it into the field that is being fuzzed. In this version of the fuzzer, the fuzzed string is inserted into the chat message body, but the fuzzer can be modified to test different fields in different parts of the Ricochet protocol.

```
import socket
import sys
import time
import os
import base64
import ricochet_pb2
import Crypto.Signature.PKCS1_v1_5
import Crypto.PublicKey.RSA
import Crypto.Hash.SHA
import Crypto.Hash.SHA256
import Crypto.Hash.HMAC
import struct

msgVersions = bytearray([0x49, 0x4d, 0x02, 0x01, 0x00])
quit = ("exitgracefully" + "\0").encode('ascii')
keyFile = "private_key.pem"
derFile = "private_key.der"
serverName = "xi7toy4inzy7nyjx.onion"

def OnionAddr(publicKey):
    h = Crypto.Hash.SHA.new(publicKey).digest()
    onion = base64.b32encode(h[:10]).decode('ascii').lower()
    return onion + ".onion"

def GetClientName():
    return OnionAddr(GetDERPublicKey())

def GetDERPublicKey():
    f = open(derFile, "rb")
    return f.read()

def MakePacket(channelId, payload):
    size = len(payload) + 4
    if (size > 65535):
        raise ValueError("payload is too long")
    if (channelId > 65535):
        raise ValueError("ChannelId is too large")
    bSize = struct.pack('>h', size)
    bChan = struct.pack('>h', channelId)
    packet = bSize + bChan + payload
    return packet

def ParsePacket(packet):
    channel = int.from_bytes(packet[2:4], byteorder='big')
    return channel, packet[4:]

def GenerateProof(clientHost, serverHost, clientCookie, serverCookie):
    key = (clientCookie + serverCookie)
    data = (clientHost[:16] + serverHost[:16]).encode('ascii')
    mac = Crypto.Hash.HMAC.new(key, data, Crypto.Hash.SHA256)
    return mac.digest()

def SignProof(proof):
    key = Crypto.PublicKey.RSA.importKey(open(keyFile).read())
    signer = Crypto.Signature.PKCS1_v1_5.new(key)
```

```
    h = Crypto.Hash.SHA256.new(proof)
    signature = signer.sign(h)
    return signature

def GenerateOpenChannelPacket(chType, chId, extensions):
    opChan = ricochet_pb2.OpenChannel()
    opChan.channel_identifier = chId
    opChan.channel_type = chType
    for e in extensions:
        opChan.Extensions[e] = extensions[e]
    ctrlPacket = ricochet_pb2.ControlPacket()
    ctrlPacket.open_channel.MergeFrom(opChan)
    payload = ctrlPacket.SerializeToString()
    return MakePacket(0,payload)

def GenerateProofPacket(chId, clientCookie, serverCookie):
    proofMsg = ricochet_pb2.AuthHSProof()
    proofMsg.public_key = GetDERPublicKey()
    proofMsg.signature = SignProof(GenerateProof(GetClientName(), serverName, clientCookie, serverCooki
→  e))
    proofPacket = ricochet_pb2.AuthHSPacket()
    proofPacket.proof.MergeFrom(proofMsg)
    payload = proofPacket.SerializeToString()
    return MakePacket(chId, payload)

def GenerateChatPacket(chId, message):
    chatMsg = ricochet_pb2.ChatMessage()
    chatMsg.message_text = message
    chatPacket = ricochet_pb2.ChatPacket()
    chatPacket.chat_message.MergeFrom(chatMsg)
    payload = chatPacket.SerializeToString()
    return MakePacket(chId, payload)

if __name__ == "__main__":
    filename = sys.argv[1]
    pid = int(sys.argv[2])

    print("Got PID: " + str(pid))
    print("starting client, reading from " + filename)

    f = open(filename, 'rb')
    afl = f.read()
    f.close()
    authChannel = 5
    chatChannel = 7
    time.sleep(1)

    sock = socket.socket()
    sock.connect(("127.0.0.1", 55555))

    sock.send(msgVersions)
    resp = sock.recv(256)

    # Open auth channel
    clientCookie = os.urandom(16)
    msg = GenerateOpenChannelPacket("im.ricochet.auth.hidden-service", authChannel, {ricochet_pb2.clien
→  t_cookie : clientCookie})
    sock.send(msg)
    resp = sock.recv(1024)
```

```
    # Check if channel opened
    channel, payload = ParsePacket(resp)
    if channel == 0:
        response = ricochet_pb2.ControlPacket()
        response.ParseFromString(payload)
        if not response.channel_result.opened:
            print("Channel did not open")
        serverCookie = response.channel_result.Extensions[ricochet_pb2.server_cookie]

    # Generate proof
    msg = GenerateProofPacket(authChannel, clientCookie, serverCookie)
    sock.send(msg)
    resp = sock.recv(1024)
    channel, payload = ParsePacket(resp)
    if channel == authChannel:
        response = ricochet_pb2.AuthHSPacket()
        response.ParseFromString(payload)
        print("Accepted: " + str(response.result.accepted))
        print("Known: " + str(response.result.is_known_contact))

    # Open chat channel
    msg = GenerateOpenChannelPacket("im.ricochet.chat", chatChannel, {})
    sock.send(msg)
    resp = sock.recv(1024)
    channel, payload = ParsePacket(resp)
    if channel == 0:
        response = ricochet_pb2.ControlPacket()
        response.ParseFromString(payload)
        if not response.channel_result.opened:
            print("Channel did not open")

    # Send a chat message with AFL payload
    msg = GenerateChatPacket(chatChannel, afl)
    sock.send(msg)

    time.sleep(1)
    sock.send(quit)
```

Finally, a launcher was written in C. This small launcher is compiled with AFL and spawns the Ricochet client, as well as the Python client that will send the fuzzed packet to Ricochet.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string>

int main(int argc, char** argv, char **arge)
{
    char *cmd[] = { "ricochet" };
    char launcher[100];
    int pid = getpid();
    sprintf(launcher, "python3.4 client.py %s %d &", argv[1], pid);
    system(launcher);
    execve("ricochet/ricochet", cmd, arge);
    return(0);
}
```

Once built, fuzzing can be started with:

```
afl-fuzz -t 25000+ -m none -i seeds/ -o output/ ./launcher @@
```

NCC Group recommends that the fuzzer continue to be run on different fields of all the Ricochet protocol messages, ideally over extended periods of time.