Instant Message Whispering via Covert Channels

Qualitätssicherungsdokument

Gruppe 35:	Jan Simon Bunten <jan_simon.bunten@stud.tu-darmstadt.de> Simon Kadel <simon.kadel@stud.tu-darmstadt.de> Martin Sven Oehler <martin_sven.oehler@stud.tu-darmstadt.de> Arne Sven Stühlmeyer <arne_sven.stuehlmeyer@stud.tu-darmstadt.de>
Teamleiter:	Philipp Plöhn <philipp.ploen@stud.tu-darmstadt.de>
Auftraggeber:	Carlos Garcia <carlos.garcia@cased.de> FG Telekooperation FB 20 - Informatik
Abgabedatum:	31.1.2014

TECHNISCHE UNIVERSITÄT DARMSTADT

Bachelor-Praktikum WS 2013/2014
Fachbereich Informatik

Inhaltsverzeichnis

1. Einleitung **2**

2. Qualitätsziele **4**
2.1. Reife 4
2.2. Testbarkeit 5
2.3. Erweiterbarkeit 5

A. Anhang **7**
A.1. Code Review check list 7

1 Einleitung

Ziel des Projekts ist es, eine Bibliothek zu entwickeln, die es ermöglicht, unentdeckt Kommunikationskanäle zu einem oder mehreren anderen Teilnehmern zu öffnen. Die Bibliothek besteht aus einem Framework und Covert Channels als Plugins (siehe Abbildung 1). Diese werden verwendet, um die Kommunikation vor Dritten zu verstecken.

Covert Channels

Covert Channels sind Kommunikationskanäle, die von außen nicht als solche erkennbar sind. In der Literatur sind viele unterschiedliche Covert Channels bekannt.

Im Unterschied zur Kryptographie, die nur die Daten eines Kanals verbirgt, wird der ganze Kanal verborgen. Dadurch wird es Dritten erschwert, die Verkehrsdaten einer solchen Verbindung (Zeitpunkt, Dauer) auszuwerten. Dazu kann ein offener Kanal zum Empfänger erstellt werden, in dem die Nachricht versteckt wird. Ist es dem Nutzer eines Covert Channels zudem möglich, die Pakete einer bestehenden Verbindung anderer Teilnehmer des Netzwerks zu verändern, kann auch dieser Kanal verwendet werden. Dadurch besteht keine direkter Kanal zwischen Sender und Empfänger, wodurch die Kommunikation noch besser versteckt ist.

Wie bei offenen Kanälen ist es auch bei Covert Channels von großer Bedeutung, wie groß der Datendurchsatz ist und wie zuverlässig die Informationen übertragen werden. Vor allem der Datendurchsatz ist bei Covert Channels üblicherweise stark beschränkt.

Implementierung

Das Hauptziel ist, ein Framework zu implementieren, das notwendige Funktionen für Covert Channels bereitstellt. Dazu gehören das Öffnen und Schließen von Covert Channels, das Senden von selbst erstellten Paketen, das Empfangen von Paketen und das Anzeigen von Statistiken der geöffneten Kanäle. Die eigentlichen Covert Channels können als Plugins hinzugefügt werden. So soll sichergestellt werden, dass das Framework für unterschiedliche Covert Channels genutzt werden kann.

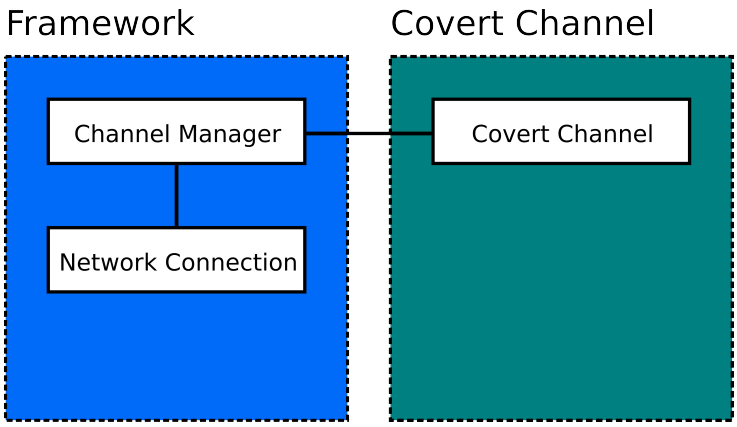

Abbildung 1.1.: Aufbau der Bibliothek

Neben dem Framework werden wir im Rahmen des Projekts drei unterschiedliche Covert Channels implementieren. Der erste, relativ einfache, Channel versteckt die Informationen im Header von TCP Paketen. Der zweite, etwas kompliziertere, Channel ist ein Timing Channel, bei dem der Sender Pakete in bestimmten zeitlichen Abständen abschickt. Der Empfänger misst die Abstände zwischen den Paketen, um die Nachricht zu erhalten. Abschließend soll ein dritter, noch zu spezifizierender, Covert Channel implementiert werden.

Das Projekt soll als Open-Source veröffentlicht werden, um es anderen zu ermöglichen, die Bibliothek in ihren Projekten zu verwenden oder sie weiterzuentwickeln.

Zur beispielhaften Anwendung werden wir ein Plugin für den Instant Messenger Pidgin[1] schreiben, welches die Funktionen unserer Bibliothek nutzt.

[1] http://www.pidgin.im

2 Qualitätsziele

2.1 Reife

Die in unserem Projekt entstandene Bibliothek soll in anderen Projekten eingebunden werden können. Andere Entwickler müssen sich darauf verlassen können, dass die Bibliothek nicht der Grund für Fehlverhalten oder Abstürze ist, sondern fehlerfrei funktioniert.

Um die Reife unseres Projekts zu erhöhen, versuchen wir möglichst viele Fehler zu finden und zu beheben. Das geschieht bei uns durch Testen und durch Code Reviews.

Zum Schreiben von automatisierten Tests in C++ benutzen wir die boost.test-Bibliothek[1]. Wir schreiben sowohl Tests für einzelne Methoden (Unittests), als auch für Funktionen der Bibliothek (Systemtests). Mit den Tests soll vollständige Statement Coverage erreicht werden, wobei Getter-, Setter- und einfache Wrapperfunktionen ausgenommen sind.

In unseren Code Reviews untersuchen wir den Code unter bestimmten Aspekten auf typische Fehlerquellen. Dabei wird der Code anhand einer Checkliste (siehe Anhang A.1) zum Beispiel auf Punkte wie Endlosschleifen oder Feldgrenzen hin überprüft.

Die Unittests werden vor oder während der Entwicklung vom Entwickler geschrieben und unseren automatischen Tests hinzugefügt. Nach der Implementierung einer User Story schreibt der Entwickler die Systemtests für die in der User Story beschriebene Funktionalität. Dabei stellt er sicher, dass die geforderte Abdeckung erreicht wird. Das Durchlaufen der Tests und die Testabdeckung werden bei den Code Reviews überprüft.

Zusätzlich werden die automatischen Tests am Ende eines Sprints auf der aktuellen Version der Software komplett ausgeführt. Auftretende Fehler werden im Ticketsystem unseres SCM-Servers eingetragen und bei der Planung der Sprints mit berücksichtigt. Das Durchführen wird mit Datum und Betriebssystem, auf dem die Tests durchgeführt wurden, in einer Liste festgehalten. Ein Teammitglied wurde dafür ausgewählt zu überprüfen, ob die Tests durchgeführt wurden.

Die Code Reviews werden am Ende jeder User Story durchgeführt. Dafür wird ein neues Ticket erstellt, wodurch alle Teammitglieder benachrichtigt werden, dass ein Code Review aussteht. Code Reviews haben Priorität und müssen innerhalb von drei Tagen durchgeführt werden. Für das Code Review trägt sich ein an der User Story unbeteiligtes Teammitglied ein. Es geht den in der User Story geschriebenen Code durch und überprüft dabei die Punkte auf der Checkliste. Wenn ein Punkt nicht erfüllt ist, wird das entsprechend vermerkt und an den Entwickler weitergeleitet. Dieser behebt die Fehler im Code und erstellt dann wieder ein Ticket für ein Review. Dieser Prozess wird solange wiederholt, bis alle Punkte der Checkliste abgehakt sind. Erst dann gilt die User Story als abgeschlossen.

[1] http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/index.html

2.2 Testbarkeit

Unsere Bibliothek soll nach Abschluss als Open-Source-Projekt veröffentlicht werden, damit andere Entwickler ihre eigenen Covert Channels hinzufügen oder die Bibliothek ihren Bedürfnissen entsprechend anpassen können. Außerdem soll sie möglicherweise in weiteren Bachelorpraktika erweitert werden. Deshalb soll es leicht möglich sein, einzelne Module der Software auszutauschen oder zu verändern.

Um dies zu vereinfachen, muss schnell feststellbar sein, ob ein neues oder verändertes Modul korrekt funktioniert. Dazu ist es nötig, dass einzelne Module leicht und unabhängig von anderen Modulen testbar sind.

Wir wollen den Aufwand ein Modul zu testen gering halten. Um dies zu bewerten, haben wir eine Checkliste mit fünf für die Testbarkeit relevanten Kriterien aufgestellt:

1. Beobachtbarkeit: Das Testergebnis kann beobachtet werden, um es mit dem erwarteten Ergebnis abzugleichen.

2. Kontrollierbarkeit: Das Testobjekt kann in den für den Test erforderlichen Zustand gebracht werden.

3. Isolierbarkeit: Das Testobjekt kann isoliert getestet werden, damit eine Abhängigkeit des Testresultats von anderen Modulen verhindert wird.

4. Trennung der Verantwortlichkeit: Das Testobjekt hat eine wohldefinierte Verantwortlichkeit, wodurch erleichtert wird, die Schnittstelle zu anderen Modulen zu testen. Die Schnittstellen beziehungsweise Verantwortlichkeiten wurden dazu im Vorfeld festgelegt.

5. Automatisierbarkeit: Die Tests lassen sich automatisieren, damit schnell ein Testergebnis vorliegt.

Die Bewertung der Testbarkeit erfolgt für jede User Story durch den Entwickler selbst, bevor er Tests schreibt. Da dies vor oder während der Entwicklung geschieht, ist es auch möglich, mehrfach eine Bewertung durchzuführen. Ist ein Punkt der Liste nicht vollständig erfüllt, ist es die Aufgabe des Entwicklers, das Design entsprechend anzupassen.

2.3 Erweiterbarkeit

Um das Benutzen und Ändern unseres Frameworks zusätzlich zu erleichtern, ist es wichtig, dass Code und Aufbau unserer Software verständlich sind. Außerdem ist beim Entwurf des Designs zu beachten, dass es modifizierbar ist. Diese beiden Punkte lassen sich unter Erweiterbarkeit zusammenfassen.

Die Verständlichkeit wird durch zwei Maßnahmen verbessert. Einerseits wird durch die Dokumentation der Funktionen und Klassen der Bibliothek die Funktionsweise verdeutlicht. Andererseits verbessern einheitliche Bezeichner und eine einheitliche Struktur des Codes die Lesbarkeit. So können sich Nutzer leichter in der Bibliothek zurechtfinden, wenn sie Änderungen oder Erweiterungen vornehmen möchten.

Auch die Modifizierbarkeit wird durch diese beiden Maßnahmen verbessert. Zusätzlich ermöglichen wir durch Modularität und abstrakte Klassen, dass Entwickler Teile der Bibliothek

leicht austauschen können. Die Aufgaben und Schnittstellen der Module und die Methoden der abstrakten Klassen werden in der Dokumentation festgehalten.

Die Dokumentation der Funktionen und Klassen wird durch den Einsatz der Software Doxygen[2] und entsprechend formatierten Kommentaren im Quellcode automatisch erstellt. Gegen Ende des Projekts (Anfang März) sind zwei Wochen vorgesehen, in denen die erzeugte Dokumentation überprüft und erweitert wird.

Eine einheitliche Bezeichnung und Formatierung stellen wir durch Code Conventions sicher. Diese wurden zu Beginn des Projekts auf Grundlage der relativ weit verbreiteten Google-Code-Conventions für C++ von uns festgelegt und in unserem Wiki festgehalten. Die Einhaltung der Code Conventions ist ein Punkt auf der Checkliste für die Code Reviews und wird entsprechend überprüft. Falls sie nicht eingehalten werden, muss, wie bei den anderen Punkten des Code Reviews auch, der Code vom Entwickler entsprechend angepasst und dann erneut überprüft werden.

Wie es in agiler Entwicklung üblich ist, werden wir die Module an die sich erweiternden Anforderungen anpassen und ihre Schnittstellen entsprechend ändern. In den zwei Wochen gegen Ende des Projektes, die für die Dokumentation vorgesehen sind, werden wir diese dann dokumentieren.

[2] http://www.stack.nl/~dimitri/doxygen

A Anhang

A.1 Code Review check list

Code Review Check list

Date:

Reviewer:
Use Case:

Description	Comment	check
Structure		
Does the solution only cover the modules task?		
Is the code well-structured, consistent in style and consistently formatted?		
Is there unreachable code?		
Are there any leftover stubs or test routines?		
Is there any redundant code?		
Are any modules excessively complex?		
Documentation		
Are the comments consistent?		
Are the comments sufficient and understandable?		
Variables and Methods		
Are all variables/methods properly defined with meaningful, consistent names?		
Are there any redundant or unused variables/methods?		
Loops and Branches		
Are all cases covered in IF-blocks?		
Are logical structures nested correctly?		
Are loop termination conditions achievable?		
Are indexes properly initialized?		
Defensive Programming		
Are indexes and pointers tested against array bounds?		
Are imported data and input arguments tested for validity and completeness?		
Are all output variables assigned?		
Is every memory allocation deallocated?		
Code Conventions		
Are the Code Conventions fulfilled?		