CISAT: Integration von sicherheitszentrierter statischer Analyse in den Enwicklungsprozess

Daniel Schreckling Martin Johns* SVS Sectoolers†
Universität Hamburg
Vogt Kölln Str. 30
22527 Hamburg, Germany
[schreckl|johns|sectoolers]@informatik.uni-hamburg.de

Zusammenfassung

Die Verwendung von Werkzeugen zur statischen Source Code Analyse mit dem Ziel Sicherheitsprobleme zu finden, scheitert oftmals an drei Faktoren: Uneinheitliche Programminterfaces, eingeschränkter Fokus einzelner Tools und mangelhafte Unterstützung der Integration solcher Tools in automatisierte Prozesse. In diesem Papier stellen wir CISAT (Combination and Integration of Static Analysis Tools) vor, einen ersten Ansatz diese Probleme zu beseitigen. CISAT ist eine Sammlung von Konventionen, Methoden und Werkzeugen, die mit dem Ziel erstellt wurden, die oben aufgezeigten Hürden abzuschwächen.

1 Motivation / Problemstellung

Statische Analyse von Programm Code ist eine verbreitete Methode um potentielle Sicherheitslücken noch während der Entwicklung aufzudecken. Im Zuge von weitreichender Forschung in diesem Feld wurden eine breite Reihe von Ansätzen und Werkzeugen entwickelt: Angefangen mit generellen Tools, die auf der syntaktischen Ebene des Quelltextes arbeiten, wie Flawfinder [8], ITS4 [7] oder RATS [4], über spezialisierte Werkzeuge, wie dem Buffer-Overflow-Finder BOON [2] oder dem Datenfluss-Tool CQual [5], bis zu Ansätzen, die ihren Ursprung in der theoretischen Informatik haben, wie z.B. MOPS [1], das zum Aufdecken von Race Conditions verwendet werden kann.

*Die Arbeit von Martin Johns wurde im Rahmen des SECOLOGIC-Projektes (www.secologic.org) mit Mitteln des Bundesministeriums für Wirtschaft und Technologie unter dem Förderkennzeichen 01 MS 503 gefördert.

†Die SVS Sectoolers sind: Christian Beyerlein, Nils Hoier, Benjamin Leipold, Moritz Jodeit und Björn Engelmann

Allen diesen Werkzeugen, die zur freien Verwendung bereitstehen, ist gemein, dass sie in der praktischen Software Entwicklung wenig bis gar keinen Einsatz finden. Dafür können eine Reihe von Ursachen identifiziert werden:

- **Enger Fokus:** Die meisten verfügbaren Werkzeuge implementieren einen spezialisierten Ansatz. Z.B. CQual [5] bietet fortgeschrittene Möglichkeiten zur Datenflussanalyse, welche ihrerseits zum Identifizieren von Formatstring-Vulnerabilities verwendet werden. Andere Schwächen, wie z.B. Buffer Overflows, können mit diesem Werkzeug aber nicht gefunden werden. Wenn ein Programmierer nun mit Hilfe von statischer Analyse möglichst viele Sicherheitslücken finden möchte, ist er gezwungen eine ganze Reihe von Tools einzusetzen.
- **Hoher Einarbeitungsaufwand:** Jedes der erwähnten Werzeuge verfügt über individuelle Aufruf-Konventionen und Ausgabeformate. Wenn mehr als ein Tool verwendet werden soll, muss jedes Tool separat erlernt werden. Dieser Umstand erschwert mit wachsender Anzahl von Werkzeugen besonders den Prozess der Evaluation und Auswahl von einzusetzenden Werkzeugen.
- **Fehlende Einbindung in integrierte Entwicklungsumgebungen (IDEs):** Software wird heutzutage selten lediglich mit Hilfe einfacher Texteditoren entwickelt. Stattdessen werden spezialisierte Programme verwendet, die den bearbeiteten Programmcode nicht nur darstellen sondern auch "verstehen". Dieses Verstehen ermöglicht den Entwicklungsumgebungen u.a. das Einfärben von Schlüsselwörtern im Programmtext oder das graphische Aufzeigen von möglichen Programmierfehlern. Die aktuell vorhandenen Werkzeuge bieten keinerlei Unterstützung für eine Integration in diese Entwicklungsumgebungen. Die individuellen Ausgabeformate sind darauf ausgerichtet menschenlesbar zu sein und sind nicht dafür vorgesehen von maschinellen Prozessen geparst zu werden[1].
- **Keinerlei Einbindung in automatisierte Prozesse:** Im industriellen Umfeld ist die eigentliche Softwareentwicklung häufig eingebunden in eine Reihe von automatisierten Prozessen. Source Code wird in Versionskontrollsystemen verwaltet, sich in Entwicklung befindliche Applikationen werden regelmäßig Regressionstests unterzogen und verschiedenste Reporte über die Entwicklung werden automatisch erstellt und versandt. Die vorhandenen statischen Analyse Werkzeuge sind aufgrund der Heterogenität ihrer Schnittstellen ungeeignet, in solche Prozesse einfach integriert zu werden.

Die zögerliche Adaption dieser Tools hat wiederum oftmals zu einem Erlahmen der aktiven Entwicklung der Werkzeuge bzw. Verbesserung der Ansätze geführt. Ein bekannter Teufelskreis in der offenen Software Entwicklung: Nur viel verwendete Programme werden weiterentwickelt aber nur Programme, die eine aktive Weiterentwicklung bieten, finden Verwendung.

2 Das CISAT Framework

In Abschnitt 1 wurden die aktuell bestehenden Gründe erörtert, die eine Adaption von sicherheitsorientierter statischer Analyse erschweren. Im Folgenden wird das CISAT (Combination

[1] Die XML-Option von RATS [4] ist hier eine löbliche Ausnahme.

and Integration of Static Analysis Tools) Framework vorgestellt. CISAT ist eine Sammlung von Konventionen, Methoden und Werkzeugen, die mit dem Ziel erstellt wurde, die oben aufgezeigten Hürden abzuschwächen.

2.1 Definition einheitlicher Schnittstellen

Die meisten der aufgelisteten Probleme haben ihre Ursache in den uneinheitlichen Schnittstellen der existierenden Werkzeuge. Aufgrund dieser Heterogenität erhöht sich der potentielle Schulungsaufwand linear mit der Anzahl der verwendeten Produkte, und die Einbindung eines Werkzeugs in automatische Prozesse muss immer auf das jeweilige Tool maßgeschneidert sein.

Aus diesem Grund führen wir im Rahmen des CISAT Frameworks eine einheitliche Schnittstelle für sicherheitszentrische statische Analyse Tools ein. Diese einheitliche Schnittstelle besteht aus zwei Komponenten: Einer standardisierten Aufrufkonvention und einem allgemeinen XML-basierten Ausgabeformat für das Scanergebnis. Bei der Definition dieser Schnittstelle wurde explizit auf die speziellen Anforderungen bestehender und eventueller zukünftiger Werkzeuge geachtet.

Im Rahmen unserer Arbeit haben wir folgende quelloffene Tools erweitert, so dass sie das CISAT Interface implementieren: Flawfinder [8], ITS4 [7], RATS [4], BOON [2], CQual [5], MOPS [1] und Splint [3].

Uniforme Aufruf-Syntax: Alle in CISAT integrierten Tools sind folgendermassen aufrufbar: `toolname <liste von Files>`. Tools, die in ihrer ursprünglichen Form eine andere Aufruf-Syntax erwarten, wurden mit Wrapper-Programmen versehen, die von dieser ursprünglichen Schnittstelle abstrahieren. Diese Wrapper-Programme haben weiterhin die Aufgabe den Aufruf mit sinnvollen default Optionen zu versehen.

Vollständiges und maschinenlesbares Ausgabeformat: Alle in CISAT integrierten Tools geben als Ausgabe ein einheitliches XML-basiertes Ergebnisformat zurück. Dieses Format wurde so konzipiert, dass es die Obermenge aller Informationstypen, die in der Ergebnismenge einer statischen Sicherheitsanalyse vorkommen, enthält. Weiterhin ist es dafür ausgelegt, Ergebnismengen, die von einer Kombination aus unterschiedlichen Tools erzeugt wurden (siehe 2.2), zusammenzufassen. Siehe Abbildung 1 für ein Beispiel.

Eine derartige einheitliche Schnittstelle hat zwei positive Eigenschaften: Einerseits kann ein Entwickler, der geschult wurde eines der Tools zu verwenden, auch alle anderen Werkzeuge, welches die Schnittstelle implementiert hat, einsetzen, ohne sich zeitaufwendig neu einarbeiten zu müssen. Andererseits können Mechanismen zur Integration statischer Sourcecode-Scanner in größere Prozesse generisch vorgenommen werden, so dass ein Auswechseln des verwendeten Werkzeugs keinen erneuten Entwicklungsaufwand verursacht. Diese Eigenschaft ist die Grundlage für die in Abschnitt 2.3 und 2.4 vorgestellten Techniken. Weiterhin können über Wrapper-Programme beliebige weitere Tools mit dem einheitlichen Interface versehen und so einfach in bestehende Prozesse eingebunden werden.

```
<?xml version="1.0"?>
  <scan-result xmlns="http://www.xyz.de/sectoolers/scan-result/0.9/">
   ...
  <files>
    <file>examples/main.c</file>
  </files>
  <tools>
      <tool>
        <name>cqual</name>
        <version>v0.991-modified</version>
        ...
      </tool>
    ...
  </tools>
   ...
  <problems>
    <problem>
      <toolname>cqual</toolname>
      <severity>1.0</severity>
      <probability>0.5</probability>
      <category>type</category>
      <location>
        <file>formatstring.c</file>
        <line>23</line>
        <field>*env</field>
        <text>incompatible types in assignment</text>
      </location>
      <trace>
        <location>
          <file>/usr/local/share/cqual/prelude.cq</file>
          <line>58</line>
          <text>$tainted &lt;= *getenv_ret</text>
        </location>
         ...
        <location>
          <file>/usr/local/share/cqual/prelude.cq</file>
          <line>33</line>
          <text>*printf_arg1 &lt;= $untainted</text>
        </location>
      </trace>
    </problem>
    ...
   </problems>
</scan-result>
```

Abbildung 1: Beispiel einer XML Ausgabe

2.2 Kombination existierender Analysewerkzeuge

Wie im Abschnitt 1 aufgezeigt, führt der oft enge Fokus der einzelnen Werkzeuge dazu, dass es nötig sein kann mehr als eines der verfügbaren Tools parallel einzusetzen. Aus diesem Grund enthält unser Framework eine erweiterbare Kombinationskomponente, die es ermöglicht einzelne Tools miteinander zu verbinden. Realisiert wurde die Kombinationskomponente mit SATEC, einer von uns konzipierten Highlevel API, die darauf ausgelegt ist, mit dem CISAT XML-Ergebnisformat umzugehen. Ein spezieller Fokus dieser API liegt auf der Algorithmik, mit der die Ergebnisse der Einzeltools zusammengeführt werden. Funktionen wie Voting, Schwellenwerte und Verschmelzen (siehe Abbildung 2) können so einfach implementiert werden. So könnten gezielt die eventuellen Schwächen einzelner Tools ausgeglichen und deren Stärken betont werden.

Die Kombinationskomponente ihrerseits implementiert ebenfalls die einheitliche Aufrufsyntax und verwendet für ihr Endergebnis das allgemeine Ausgabeformat, was dazu führt, dass die Komponente von außen betrachtet sich exakt wie ein generisches Scanner-Tool verhält. Dieses erlaubt beispielsweise den Einsatz von kombinierten Werkzeugen in den in Abschnitt 2.3 und 2.4 vorgestellten Mechanismen.

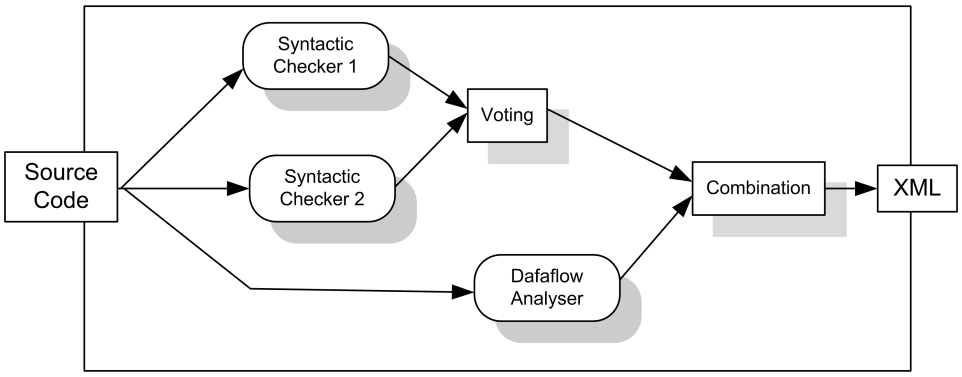

Abbildung 2: Schematische Ansicht der Kombinationskomponente

2.3 Einbindung in Versionen-Kontroll-Systeme

Zur Archivierung, Distribution und vorallem zur Versionierung während des Entwicklungsprozesses großer und kleiner Software Projekte werden Versionen-Kontroll-Systeme (VCS) eingesetzt. Ihre Funktionalität ermöglicht die verteilte Bearbeitung eines Projektes. In Unternehmen kann Software so zentral archiviert und der Entwicklungsprozess entsprechend kontrolliert und verwaltet werden. Auch in der OpenSource Gemeinde erfreut sich dieses Werkzeug großer Beliebtheit. Entwickler können das Fortschreiten eines Projektes zu unterschiedlichen Zeiten mit unterschiedlichem Wissensstand unterstützen.

Durch den Einsatz dieser Systeme bei der Softwareentwicklung ist es natürlich wichtig, dass Änderungen des Source Codes nicht zu möglichen Verwundbarkeiten eines Projektes führen. Eine automatische Prüfung ist aufgrund der häufig umfangreichen Änderungen unumgänglich.

Aus diesem Grund ermöglicht CISAT auch die transparente Integration beliebiger statischer Analyse Werkzeuge in existierende VCS.

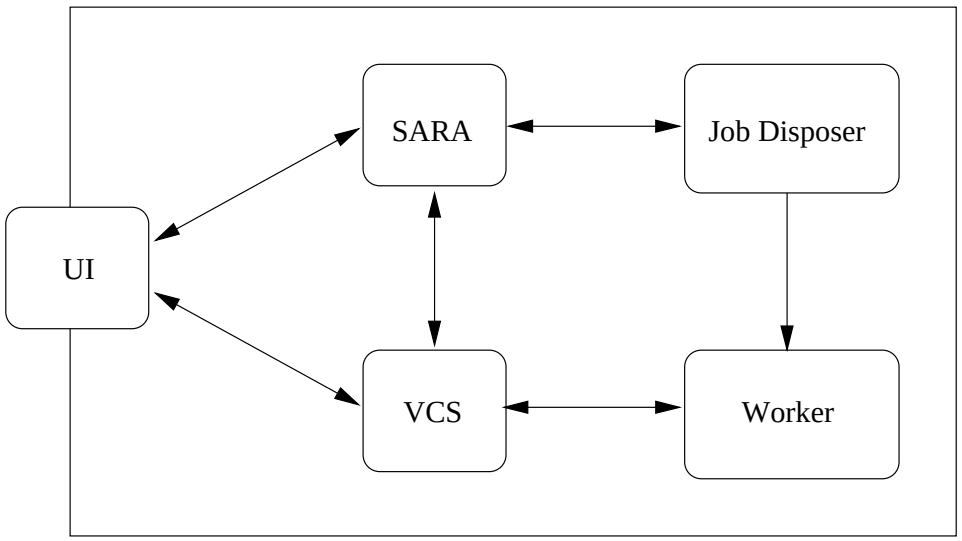

Abbildung 3: CISAT in einem VCS

2.3.1 Aufbau

Abbildung 3 zeigt den schematischen Aufbau dieser Integration. Bei der Implementation wurde besonderer Wert darauf gelegt, dass nahezu keine Veränderungen des verwendeten VCS notwendig sind. Eine transparente Integration unseres Frameworks in existierende Versionenkontrollsysteme läßt sich in den meisten Fällen durch einfache *commit hooks* realisieren. Sie machen eine Anpassung der VCS Clienten hinfällig.

Die fünf interagierenden Komponenten der VCS Integration sind das *VCS* selbst, das *User Interface*, *SARA*, der *Job Disposer* und die einzelnen *Worker*. Diese Komponenten werden im Folgenden kurz erklärt. Ihre Interaktion mit anderen Teilen des vorgestellten Frameworks werden in Abschnitt 2.3.2 näher erläutert.

Wie bereits erwähnt soll die Integration der vorgestellten Mechanismen in bestehende Versionierungs Systeme leicht und transparent sein. Da nahezu alle geläufigen Systeme sogenannte *Hooks* anbieten und diese keine Modifikation der Kernkomponenten eines VCS erfordern, wurden Hooks verwendet, um unserer Komponenten anzubinden. Dadurch bleibt die VCS Komponente unseres Frameworks sehr generisch und es wird nur die Implementation einer primitiven, VCS-spezifischen API notwendig, die die erforderlichen Funktionalitäten in Abschnitt 2.3.2 zur Verfügung stellt. In unserer momentanen Implementation unterstützen wir das Versionskontrollsystem git [6].

Die *User Interface* Komponente ist ebenfalls generisch gehalten und repräsentiert sowohl IDE Plugins als auch rudimentäre Kommandozeilen Interfaces. Sie ermöglichen die Kommunikation des Nutzers mit der Komponente SARA als auch indirekt mit dem VCS. Für spezifische Applikationsszenarien verweisen wir wiederum auf Unterkapitel 2.3.2.

Die wichtigsten Funktionalitäten unserer VCS Integration übernimmt *SARA*, der *Static Analysis Result Administrator*. Je nach Konfiguration erlaubt SARA beispielsweise die Konfliktauflösung im Falle widersprüchlicher Prüfergebnisse, benachrichtigt entsprechende Nutzer über mögliche Fehler, entscheidet aufgrund von Informationen des VCS und intern gespreicherter Daten, welche Module eines Projektes auf mögliche Schwachstellen geprüft werden sollen und stellt schließlich einem Nutzer relevante Prüfergebnisse zu seinem Projekt oder seiner Datei zur Verfügung.

Der *JobDisposer*, die vierte Komponente dieser Integration, übernimmt eventuelle Prüfaufträge, die SARA generiert hat. Inhalt und Format der Aufträge, auch `Jobs` genannt, werden genauer in 2.3.2 erläutert. Diese Aufträge auf *Worker* zu verteilen, die dem Server zur Verfügung stehen und die auch die entsprechenden Anforderungen für die Abarbeitung eines Jobs erfüllen, ist die Aufgabe des Disposers. Ähnlich den bereits beschriebenen Komponenten wurde auch beim JobDisposer Wert auf ein generisches Design gelegt. Dies hat dazugeführt, dass der JobDisposer ein eigenständig arbeitender, vielseitig einsetzbarer Batch Processor ist. Vorteil des generischen Designs ist natürlich die Möglichkeit das Framework auf beliebig komplizierte bzw. einfache Analyse Werkzeuge anzupassen und die Rechenleistung entsprechend komplexer Netzwerke effektiv zu nutzen.

Worker führen durch den Einsatz, der durch Wrapper modifizierzten, Statischen Analyse Werkzeuge den eigentlichen Prüfvorgang des entsprechenden Source Codes durch, indem sie die vom JobDisposer zugeteilten `Jobs` ausführen. Prinzipiell können Worker als Hosts eines Netzwerkes angesehen werden, die einem Server, dem JobDisposer, ihre *Fähigkeiten* mitteilen und

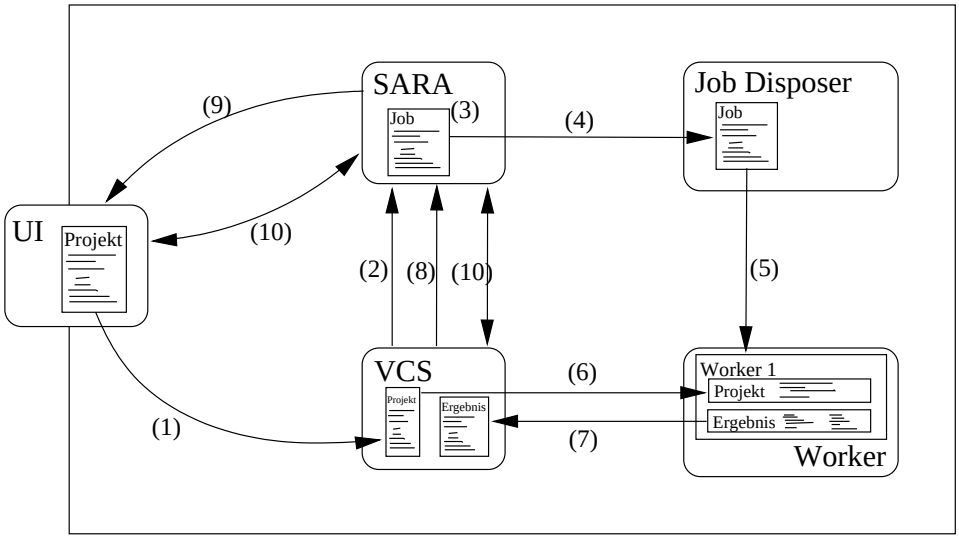

Abbildung 4: Funktionsweise von CISAT

auf entsprechende Aufträge warten. *Fähigkeiten* sind in unserem Fall die zur Verfügung stehenden Analyse Werkzeuge. Hier sei zu erwähnen, dass natürlich auch mehrere Worker auf einem Host existieren können.

2.3.2 Funktionsweise

Nachdem wir im letzten Abschnitt die einzelnen Komponenten kurz erläutert haben möchten wir im Folgenden etwas genauer auf Ihr Zusammenspiel eingehen. Hierbei setzen wir voraus, dass der Leser mit der prinzipiellen Funktionsweise eines VCS vertraut ist und dessen Aufbau verstanden hat. Deshalb erklären wir Vorgänge wie das Aus- und Einchecken, Mergen, Commiten und ähnliche Operationen eines VCS nicht. Vielmehr beschränken wir unsere Beschreibung auf einen generischen Use-Case des Frameworks. Aus Platzgründen verzichten wir hier auf eine ausführliche Beschreibung der einzelnen Workflows.

Die im Folgenden näher erklärten Schritte beziehen sich auf Abbildung 4, in der sie ihre grafische Repräsentation finden.

Sobald der Entwickler auf übliche Art und Weise eine neue Version seines Projektes bzw. seiner Dateien in das VCS überträgt (1), beispielsweise durch ein `commit`, initiiert ein VCS Hook die folgenden, für den Nutzer transparenten, Schritte.

Der VCS Hook informiert SARA (2) über das Vorhandensein einer neuen Version eines Projektes. Auf seiner Konfiguration basierend generiert SARA nun einen `Job` (3) für den neuen Commit. Auch zur Beschreibung eines Jobs bedienen wir uns XML (siehe auch Abbildung 5). Ein Job enthält im groben die Schritte, die ein Worker abzuarbeiten hat. Somit ist spezifiziert, wo der zu prüfende Source Code abgelegt ist, welche Analyse Werkzeuge zu verwenden sind, und wo die Prüfergebnisse zu speichern sind. Da die Verteilung der Jobs möglichst transparent geschehen soll und dies per Design nicht die Aufgabe von SARA ist, könnten die erzeugten Jobs nun direkt an den JobDisposer weitergegeben (4) werden. SARA führt allerdings noch zusätzliche Prüfungen durch, die hier kurz erklärt werden sollen.

```
<?xml version="1.0"?>
<job>
   <requires name="CISAT"/>
   <task name="VCS:git:checkout" version="0.1">
      <![CDATA[<checkout repos="/git/projekt" branch="master"
                commit="9a7f84f42038dabe0ebbc9336828a"/>]]>
   </task>
   <task name="boon" version="0.1">
      <![CDATA[boon]]>
   </task>
   <task name="VCS:git:commit" version="0.1">
      <![CDATA[<commit repos="/git/projekt" branch="results">
                  <file from="boon"
                        to="master/9a7f84f42038dabe0ebbc9336828a/toller-job/boon"/>
               </commit>]]>
   </task>
</job>
```

Abbildung 5: Beispiel einer Job Beschreibung

```
<?xml version="1.0"?>
<worker id="446bb49f6e6b38fd40d27c45ec25baa0">
   <property name="CISAT"/>
   <value name="HD:Space">9396224</value>
   <plugin name="VCS:git:checkout" version="0.1"></plugin>
   <plugin name="VCS:git:commit" version="0.1"></plugin>
   <plugin name="boon" version="0.1"></plugin>
</worker>
```

Abbildung 6: Beispiel einer Worker Beschreibung

So verwaltet SARA eine interne History. Diese enthält Informationen darüber, welche Jobs dem JobDisposer weitergereicht wurden, wann ein Job dem JobDisposer weitergegeben werden soll und ob es bei der Auswertung Probleme gab. Die History stellt Methoden zur Verfügung, die es erlauben Overhead zu erkennen und zu vermeiden. Beispielsweise müssen Dateien oder Projekte, für die bereits ein Prüfergebnis vorliegt nicht nochmals geprüft werden. Dieses Vorgehen klingt zunächst einleuchtend kann aber mit fortgeschrittenen Analyse Werkzeugen, die inter-prozedurale Analyse unterstützen, zu Problemen führen. Die Unabhängigkeit von Prüfergebnissen einzelner Dateien gegenüber Änderungen in anderen Dateien ist dann nicht mehr gegeben. Diese Beobachtung ist Gegenstand unserer Forschung und führt dazu, dass bei Prüfvorgängen, die MOPS [1] einschließen, standardmäßig das gesamte Projekt geprüft wird.

Wir gehen nun davon aus, dass SARA einen Job an den JobDisposer weitergegeben hat (4). Der JobDisposer hält eine Liste an Workern vor, die CISAT zur Verfügung stehen. Jeder dieser Worker besitzt eine sogenannte Worker-Beschreibung (siehe auch Abbildung 6), die zum einen genau spezifiziert, welche Analyse Tools auf dem Worker installiert sind und andererseits angibt, welche VCS unterstützt werden. Diese Informationen werden beim initialen Anmelden eines Workers am JobDisposers initialisiert. Der JobDisposer verteilt nun die Jobs auf die einzelnen Worker (5).

Diese extrahieren zunächste den Code aus dem VCS (6), in dem das zu analysierende Projekt gespeichert ist. Sobald die Statische Analyse des Codes beendet ist, übertragen sie Ihr Prüfergebnis in das VCS, aus dem sie auch den Code extrahiert haben (7). Um von den verwendeten Analyse Werkzeugen unabhängig zu sein, soll hier nochmals betont werden, dass die Analyse Werkzeuge natürlich die in 2.1 definierten einheitlichen Schnittstellen verwenden.

Mit Abschluß des Prüfvorgangs erhält SARA eine Nachricht, dass der Job beendet wurde (8). Basierend auf den intern gespeicherten Datenstrukturen und den Prüfergebnissen aus dem VCS zu diesem Job kann SARA auch weiterführende Analysemethoden durchführen, die bei-

spielsweise die richtigen Nutzer über sicherheitskritischen Code informieren (9). Im folgenden Abschnitt 2.3.3 beschreiben wir einige Probleme mit diesem Ansatz.

SARA beherrscht aber auch Standardmechanismen, die zur Benachrichtigung verwendet werden können. So kann ein Nutzer, der seine Dateien in das VCS übertragen hat, lediglich über das Ende der Prüfroutine informiert werden. Mit dem Nutzerinterface seiner Wahl, dass sich hierbei der Funktionalität von SARA und des in Abschnitt 2.1 beschriebenen CISAT XML Formates bedient (10), kann er die vorliegenden Prüfergebnisse analysieren oder gegebenenfalls Korrekturen vornehmen. In nachfolgenden Versionen soll der Nutzer auch in der Lage sein aufgetretene Falsch Positive zu markieren, um die angezeigten Fehlerlisten zu reduzieren (siehe auch Abschnitt 2.4).

2.3.3 Probleme

Obwohl SARA schon eine nicht zu vernachlässigbare Komplexität aufzeigt, ist diese Komponente noch nicht ausgereift und Gegenstand aktueller Forschung. In diesem kurzen Abschnitt seien drei Probleme aufgezeigt, die es in späteren Versionen zu lösen gilt.

Wie bereits erwähnt, verwendet SARA eine History, die nur unter der Annahme funktioniert, dass nur intra-prozedural Analyse Werkzeuge zum Einsatz kommen. Der Einsatz interprozeduraler Analyse Werzeuge kann bisher nur durch entsprechende Sonderbehandlung dieser Werkzeuge stattfinden. Dies ist natürlich nicht wünschenswert und soll in zukünftigen Versionen behoben werden. Durch mächtigere Analysewerkzeuge ergeben sich aber auch neue, interessante und forschungsrelevante Fragestellungen, die es zu lösen gilt.

SARA soll zukünftig nicht nur die beteiligten Entwickler über das Vorhandensein der Prüfergebnisse informieren, sondern explizit Einzelne über ihre Fehler informieren. Insbesondere bei VCS Systemen ist dies ein nicht einfaches Vorhaben. Mehrere Entwickler können Änderungen vornehmen, die isoliert betrachtet keine Sicherheitslücken enthalten. Allerdings können zwei oder mehr kombinierte Änderungen in einem Projekt sicherheitsrelevante Fehler verursachen. Interessante Frage an dieser Stelle ist, welcher Entwickler über den Fehler informiert wird bzw. wer aufgefordert wird, den entsprechenden Code zu korrigieren.

Erlaubt man dem Nutzer zusätzlich aufgetretene Falsch Positive zu markieren, um sie beim nächsten Check zu ignorieren, stellt sich eine ganz ähnliche Frage. Kann SARA in späteren Versionen Fehler ignorieren, d.h. keinen Report an einen Entwickler senden, wenn ausschließlich Fehler auftauchen, die bereits als Falsch Positive identifiziert wurden. Änderungen an einer Datei in einem VCS können im Source Code neue Fehler erzeugen, die bereits identifizierte Falsch Positive überdecken, d.h. in diesem Falle würden Fehler ignoriert, die eigentlich wieder aktuell sind, weil SARA den Entwickler über diese Fehler nicht informieren würde.

SARA, die zentrale Komponente der VCS Integration von CISAT, wirft einige sehr interessante Fragen auf, die es noch zu lösen gilt. Damit zeigen wir, dass CISAT noch nicht ausgereift ist aber um so mehr Raum für weitere interessantere Fragestellungen und deren Lösungen läßt.

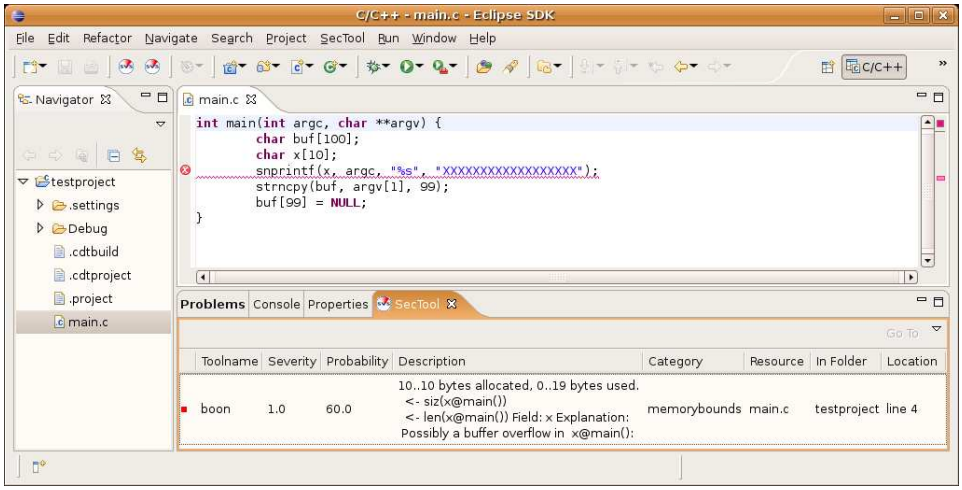

Abbildung 7: Beispielhafte Einbindung in eine IDE

2.4 Einbindung in Entwicklungsumgebungen

Aufgrund des machinenlesbaren Ergebnisformats (siehe auch 2.1) erlaubt unser Framework auch eine einfache Integration in bestehende Entwicklungsumgebungen. Hierzu ist lediglich ein Plugin notwendig, welches die Schnittstellen dem Nutzer geeignet zugänglich macht. So kann beispielsweise die Tool-Ausgabe so aufbereitet werden, dass problematischer Code direkt in der IDE markiert und eine eventuelle Problemanalyse angezeigt wird.

Ein solches Plugin kann auf diese Weise eine IDE um folgende wichtige Eigenschaften erweitern:

- **On-Demand Checking:** Falls ein Entwickler modifizierten Code auf Schwachstellen prüfen möchte, kann er bei Bedarf während der Entwicklung ein geeignetes Tool wählen, was seinen Code auf eventuelle Fehler prüft.
- **Project Analyse:** Bei größeren Projekten ist eine on-demand Lösung praktisch nicht mehr durchführbar. Hier bietet das CISAT Framework die Möglichkeit das Projekt in einem, wie in 2.3 beschriebenen, VCS zu versionieren und nach Fertigstellung die entsprechenden Prüfergebnisse in der IDE anzeigen zu lassen. Die Werkzeuge, die bei dieser Prüfung zum Einsatz kommen sollen, sind hierbei durch den Nutzer konfigurierbar.

Dank der eingeführten, vereinheitlichten Schnittstellen-Definition kann ein derartiges IDE-Plugin für jedes der unterstützten Tools verwendet werden, ohne speziell angepasst werden zu müssen. Im Rahmen unserer Implementierung haben wir eine erste Integration in die Eclipse-IDE vorgenommen (Siehe Abb. 7).

3 Zusammenfassung

In diesem Papier präsentierten wir CISAT, ein Framework zur Vereinheitlichung, Kombination und Integration von sicherheitszentrischen statischen Analyse Tools. Wie wir dargelegt

haben, ermöglicht eine derartige Standardisierung und Zusammenfassung von verschiedenen Tools die effiziente und zielgerichtete Einbindung von statischer Analyse in den industriellen Software-Entwicklungsprozess. Eine erste Implementierung unseres Frameworks wurde von der Gruppe „SVS Sectoolers" an der Universität Hamburg durchgeführt und steht unter einer open-source Lizenz zur Verfügung. Somit kann diese Implementierung die Basis bieten, weitere fortgeschrittenere Prozesse zu realisieren.

Literatur

[1] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, 2002.

[2] D.Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of NDSS 2000*, 2000.

[3] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilites. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[4] Inc. Secure Software. Rats - rough auditing tool for security. Tool, <http://www.securesoftware.com/resources/download_rats.html>, 2001.

[5] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[6] Linus Torvalds and Junio Hamano. git. Tool, <http://git.or.cz/>, 2006.

[7] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, 2000.

[8] David A. Wheeler. Flawfinder. Tool, <http://www.dwheeler.com/flawfinder>, 2001.