Source Code Analysis technologies have significantly evolved in recent years - making improvements in precision and accuracy with the introduction of new analysis techniques like flow analysis. This article describes this evolution and how the most advanced capabilities available today like query-based analysis and Knowledge Discovery can be leveraged to create a platform for Application Risk Intelligence (ARI) to help implement a proactive security program.

As attacks become more financially motivated, and as organizations get better at securing their network, desktop and server infrastructure, there has been a steady shift in cyber attacks to the application level. To address these new risks, several technology markets for application security have emerged including what Gartner calls Static Application Security Testing (SAST).

Can we really use Source Code Analysis (SCA) to detect vulnerabilities in a way that will give us satisfactory risk assurance? The answer varies; we are able to find problems with sufficient accuracy if they are manifested in commonly known sequences like a SQL injection. Many flaws however are presented in the specific implementation of the applications business logic¹, and adopting the standard analysis to handle these proprietary sequences make the task almost impossible. The reason is that current technologies, even those employing flow analyses, are based on “black-box” implementation and therefore can only handle known sequences.

A probable detection method of suspicious sequences can be through the identification of an irregularity within a set of common sequences. Enabling detection of proprietary and stochastic sequences requires implementation of an open platform for application understanding, which not only finding basic vulnerabilities but also provides systematic application risk intelligence.

The key enabler of such a platform is the exposure of all application building blocks like data structure and flow using an abstractive model common to all languages using an abstractive model common to all languages and storing the wealth of know-how in a persistent store like data base. Once the application is exposed we must provide tools to model the known and proprietary sequences as well as exposing irregularities in the common sequences. In this article, we define the required solution elements. It concludes by presenting reference architecture to such a platform.

---

¹ OWASP (Open Web Application Security Project) is considered as the focal point for all application-security related information. http://www.owasp.org/index.php/Businss_logic_vulnerability

---

Maty Siman

is a technology and security expert. He founded Checkmarx in 2006 to share his innovations in application security based on experience he gained in securing intelligence systems.

E-Mail: maty@checkmarx.com

---

² There are several implementations of CodeDom. The one used by Microsoft is the closest in meaning to the CodeDom developed by Checkmarx and used in this article. http://msdn.microsoft.com/en-us/library/system.codedom.aspx
changing a Boolean value from True to 

analysis, in order to understand how

A query can be used to conduct impact

by using simple query form.

enables true correlation between the two

the same database as the CodeDom, and

of these graphs, allows us to store them in

statement. Formal and unified modeling

termines the effect of a certain condition

ue affects the application behavior and de-

can reveal for example how a specific val-

of the dynamic flows of the application. It

ly). It enables tracing and understanding

Graph (DFG, CFG and CDG – respective-

Flow Graph and Control Dependency

Flow Graph, Control Flow Graph and Control Dependency

(DFG, CFG and CDG – respective-

ly). It enables tracing and understanding

of the dynamic flows of the application. It

can reveal for example how a specific val-

ue affects the application behavior and de-

termines the effect of a certain condition

statement. Formal and unified modeling

of these graphs, allows us to store them in

the same database as the CodeDom, and

enables true correlation between the two

by using simple query form.

Example

It becomes easy to find dead-methods

(methods that are never called). We need

to write a query that looks in the database

for all method declarations that do not

have a matching invocation statement.

The exact query will be shown in the next

chapter.

On top of the CodeDom, which represents

static properties of the code, it is possible
to compute the Data Flow Graph, Control

Flow Graph and Control Dependency

Graph (DFG, CFG and CDG – respective-

ly). It enables tracing and understanding

of the dynamic flows of the application. It

can reveal for example how a specific val-

ue affects the application behavior and de-

termines the effect of a certain condition

statement. Formal and unified modeling

of these graphs, allows us to store them in

the same database as the CodeDom, and

enables true correlation between the two

by using simple query form.

Example

A query can be used to conduct impact

analysis, in order to understand how

changing a Boolean value from True to False, affects the course of the application.

Query Language

Once we have a persistent OODB filled

with all the code properties of a scanned

application, it is possible to flexibly query it and search for any code sequence that’s

either known or proprietary. The syntax

of the query language shown here is simi-
lar to C#, with added commands and data types, such as CxList. It represents an ar-

ray of CodeDOM/Flow elements.

DOM Queries

To demonstrate what can be achieved using

a query language, we must first write a

query that finds uncalled methods. Then

we find all method declarations and put it

in a list, finally we remove all declarations

from this list that have been called with a

matching method-invocation statement.

// Find all Method Declaration in the code
CxList MethodDeclaration = All.

FindByType(typeof(MethodDecl));

// Find all Method Invocations in the code
CxList MethodInvokes = All.FindBy

Type(typeof(MethodInvokeExpr));

// Find the matching declarations of the

// invocation we found
CxList DeclarationOfInvokes = All.

FindAllReferences(Method

Invokes);

// Get the “dead” declarations
Return MethodDeclaration –

DeclarationOfInvokes;

Flow Queries

We can take it one step further and use the

DFG information and conduct an impact

analysis by finding all the places which will

be affected by changing the initialization

value of a variable called “s”.

// Look for the place where “s” is declared
CxList DeclaratorOfS = All.

FindByField(Declarator).

FindByType(typeof(Declarator)).

FindByShortName(“s”);

// Find all places which are “data-influ-

enced by”
the value assigned to “s” at its declaration
Return All.DataInfluencedBy

(DeclarationOfS);

The database can provide as a result the full

reasoning of the impact, tracing it from

source to target and backward (Fig. 1).

Security-Related Queries

As we have seen, it is possible to “ask” the

database any question about the code, ei-
	her static or tracing flow properties. A

subset of these questions is security related

and based on commonly known security

sequences. In order to find SQL Injection5

vulnerabilities we trace it in the application

using the following simple query that looks

for all the database-access in the code and

is directly influenced by the user’s input

without being sanitized properly:

// Find all places where a database is ac-

ceded
CxList db = Find_DB();

// Find all interactive user inputs
CxList inputs = Find_

Interactive_Inputs();

// Find places where user input is sanitized
CxList sanitized = Find_

Sanitize();

// Return all database access which are

influenced by an input but not sanitized
Return db.InfluencedByAndNot

Sanitized(inputs, sanitized);

We can continue with the second example.

In web applications, static variables are

shared among all users that access the sys-
tem simultaneously. This means that if a

value of a static variable is affected by a user’s input, it may be overwritten by other

users. This results in race condition. To

trace such a sequence we can use a query

that finds all the places where the static

variable is influenced by the user input:

// Find all static variables
CxList statics = All.FindAll

References(All.FindByField

Attributes(Modifiers.Static));

3 http://odbms.org

4 There were a few attempts to formalize a query

language for OODB. Most notable is OQL from ODM:


doc/p353e/xf1133402.html In this article we chose to

use CxQL as defined by Checkmarx as it better

suites code analysis techniques.

5 http://www.owasp.org/index.php/SQL_Injection

Figure 1 | Flow-graph reasoning retrieved from database

![Flow graph reasoning retrieved from database](image-url)
As said, many security-related issues are manifested in commonly known sequences and modeling is therefore generic. The platform provider should supply out-of-the-box templates to discover these sequences. This is true for other applications leveraging the open platform like: Coding standards enforcement, Quality assurance and more.

**Business Logic Flaws**

This brings us to an even more challenging aspect – the Business Logic Flaws which are usually manifested in proprietary code sequences. The examples we saw previously are considered to be “technical vulnerabilities” since they might affect whatever the actual business is that’s served by the application i.e. a book store, bank or an internet provider. With “Business Logic Flaws” it is possible to find vulnerabilities that are specific to the business process supported or enabled by the application. Although this sets a higher challenge bar, it is possible to model many of these sequences using the query language.

A common functionality in shopping carts used by any online store is the ability to change the quantity of items for purchase. In order to calculate the total amount to be paid, the system has to multiply the quantity of items the user wishes to purchase with the unit price of the item. A scenario which is often overlooked occurs when a user types in a negative quantity of items to be purchased. In such case, when an appropriate condition is not properly set, the total amount might be drastically lower than the true value, or even negative. Usually the best way to avoid this is by simply adding in the application a condition that verifies the quantity is greater than zero. In most cases such edits will prevent the risk but if the edits are neglected they can cause a fraudulent event. We can model this using a query which finds all variables called “quantity” or similar, which are influenced by user input and sent directly to the database, but their value is never checked to be positive.

A skeleton of such query might look like the following:

```csharp
// Find all variables that hold a quantity.
CxList qty = All.FindByRegex("q.t.y");

// All quantities that are influenced by user input
CxList inp_qty = qty.DataInfluencedBy(Find_Interactive_Inputs());

// All Binary Expressions that compare Qty with 0
CxList checking = All.FindByType(typeof(BinaryExpr).DataInfluencedBy(qty).DataInfluencedBy(All.FindByName("0")));

// All DB that are set from the above Qty vars
CxList db_inp_qty = Find_DB().DataInfluencedBy(inp_qty);

// Return all DB that are set user controllable Qty which is never compared to 0
Return db_inp_qty - db_inp_qty.ControlInfluencedBy(checking);
```

The example above, although schematic and specific to the described scenario, can be used as a template to any business case in which there is a need to make sure input from user is non-negative (number of students in a course, number of travelers in a flight, number of tickets to buy). Merely the Regex pattern should be changed.

Another example of a business logic-flaw can be found in an online store where it is essential to maintain tenant level privacy. It is imperative to find all the places where a customer might have access to other customers’ orders. Modeling it into query might look for all SQL statements which are set from the above Qty vars:

```csharp
// Find all DB that are set from the above Qty vars
CxList db_inp_qty = Find_DB().DataInfluencedBy(inp_qty);

// Return all DB that are set user controllable Qty which is never compared to 0
Return db_inp_qty - db_inp_qty.ControlInfluencedBy(checking);
```
that Select values from the "T_Orders" table, where the "Where" clause is not influenced by the current user ID. Again, this can be used as a template query for similar business scenarios.

**Application Understanding**

The queries discussed so far were made possible thanks to the fact that code information was stored in a query-able database. It is also possible to take further advantage of the database storage and extend the discovery to finding new vulnerabilities and better understanding the existing ones.

**Abstracting Vulnerabilities**

When applying a query on a code base, each result represents a single security breach or a business risk. Although it provides great value to find these issues, it is possible to take it to a higher abstraction level, in such a way that the results will be presented in a graphical way and reveal the correlation between results. This approach proved to be useful in better understanding the code at hand, and finding the best way to fix vulnerabilities.

A sample query that models the class hierarchy will be simply:

```csharp
// Look for the place where "s" is declared
CxList BaseClass = All.FindByType(typeof(ClassDecl));
Return = All.InheritsFrom(BaseClass);
```

And the graphical representation of the results will look like a class hierarchy (Fig. 2).

A more sophisticated query that requires 3D modeling capabilities models the CFG on the XY axis, and the call stack on the Z axis - the currently watched function is the closest CFG, and the called functions can be seen farther in the graph. Usually CFG is modeled as 2D graph that shows the flow of a function (or an application if we deal with interprocedure CFG). Call graph is also modeled as 2D graph where each function is a vertex and function call is an edge of the graph. We can combine both graphs to a single, 3D graph so the developer can see closer the CFG of the function she is interested in, and farther away the called function, so at the same time she can focus on the function at hand as well as have a quick glance on the called function.

Taking this to debugging activities, a developer has to choose whether he wants to "Step-Over" or "Step-Into" a function call. Using this graph, the developer can virtually do both at the same time – see the called code, without losing the context of the calling function.

In Fig. 3, the CFG of the application can be seen clearly, where function calls are places farther on the Z axis. Employing these graphical capabilities to the Risk Intelligence realm reveals interesting correlations and leads to accurately identifying the "Best-Fix-Location". Looking for SQL Injection in an open-source application led to 10 findings. Instead of taking care of the 10 individual results, modeling these into one graph...
(Fig. 4) shows clearly how these issues relate to each other:

It can be seen that there are very few offending input commands in the system (marked in gray), and most of the vulnerabilities pass through a single junction\(^8\) (stripes), which might be a good place to consider putting in place input-validation mechanisms as well as sanitizing the user data.

\(^8\) Basically, this can be found using max-flow/min-cut algorithm: www.sce.carleton.ca/faculty/chinneck/po/Chapter9.pdf

**Code Mining**

The fact that the code is no longer a mere text file but rather an actual information source stored in an OODB permits us to perform KDD\(^9\) (Knowledge Discovery in Databases) techniques. KDD unveils interesting previously unknown sequences in general and security vulnerabilities in particular.

For general coding practices, it can be assumed that the majority of code developed in a corporate complies with the best-coding practices defined there, and it is desired to find the places where the code doesn’t adhere to these standards. Although every coding standard can be easily written as query, the Code Mining techniques provide a method to automatically define these “MetaQueries”.

For example, a common good development practice is to have at least one statement within Catch block, and not to leave it empty. In its query form it will look for all „Catch‟s in the code („AllFindByType (typeof(CatchStmt))‟) and out of those, find the ones that their catch.statements.count property equals to 0. Using the Code Mining technique, the system automatically identifies that the property catch block in most cases in the code is not empty, and flags violations by marking the out-of-sequence entries.

A more common security-oriented example is Authentication-Bypass. After a user has successfully authenticated to the system, each subsequent page should make sure the request is through a authenticated user. Otherwise, a malicious user can go directly to that page without authenticating first, and thereby completely bypassing the authentication mechanism. This has to be done on each page that is considered as sensitive\(^10\). Often this is done correctly, but from time to time a developer neglects to put this security measure in place. As demonstrated previously, it is easy to formulate a query which makes sure that every sensitive page is correctly prefixed with “IsAuthenticated” statement. Using Code-Mining (specifically, Code-Sequence-Mining), there is no need to actually write any query. The system automatically correlates between specific statements (which are identified as sensitive, such as DB or file access) and the “IsAuthenticated” statement that most of the time appears previously.

\(^9\) The field of KDD (Knowledge Discovery in Database) is very large. One of the most cited articles in this field: [FPSS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. “The KDD process of extracting useful knowledge from volumes of data.” http://wang.ist.psu.edu/course/05/IST597/papers/Fayyad_1996.pdf

\(^10\) Microsoft Hotmail suffered from this type of vulnerability in 2002, where hackers were able to bypass security questions that users must answer before resetting their password. http://seclists.org/isn/2002/Feb/54
Use Case - Modular Scanning of Large Application Platform

To illustrate the abilities provided by application understanding, we can better demonstrate it by describing a solution to a complex analysis problem solved by using a deep understanding of the application and flows. Many modern application platforms including cloud-based offerings are multi layered including engine, platform and associated applications. The inherent problem with such monolithic platforms is that exploring vulnerabilities within a specific application or a platform module might require full platform analysis. This is not realistic especially when trying to implement a true SDLC. To enable on demand vulnerability detection of a certain module or an application, a modular scanning solution must be applied. Such a solution is impossible using conventional methods due to inter dependencies between modules. How can we leverage application understanding as to devise the solution?

We start with an example (Fig. 5). How might a SQL Injection be manifested in an application riding on a platform? Looking for SQL Injection discovers that Module1, at the application layer, connects to Module5 at the platform layer, which in turn is connected to Module7 at the engine layer, in such a way that input from the user at Module1 finds its way to Module7 without being cleaned properly. It is obvious that in order to find the SQL Injection, all three layers have to be scanned. However, there are many applications that ride on the platform, and we do not want to scan the entire platform each time we change the application. Furthermore, the source of the platform is mostly unavailable to the application developers.

Using application understanding techniques we can explore the platform using “mapping” queries, and create “plugins” – the “essence” of the platform. (Fig. 6) The plug-ins keep the inter-dependencies information and that will come in handy later when scanning individual modules.

A plug-in might include information like: function foo() in Module5 is the part of the platform’s API and it connects to Module7 that accesses a database and includes no sanitation; hence it’s exposed to database manipulation. Consequently all application calls to Module5 for data base access cannot rely on the called module to be safe and should sanitize date prior to the call.

After creation of plug-ins, only the individual application needs to be scanned. The system automatically solves the linkage between the modules and determines which of the linked modules are safe or not.

Once the platform is “understood” we can safely perform modular scanning, detecting the vulnerabilities in a single module without losing accuracy due to inter dependencies.

Reference Architecture

In order to build an effective and flexible platform which enables true application understanding and risk intelligence, the developers should follow some architectural principals laid out in the following reference architecture (Fig. 7).

- The most important principals are as follows:
  - Conversion to Common language form
  - Generate DOM and Flow properties
  - Store in persistent a database
  - Expose the abstractive model
  - Enable data access using a formal query language
  - Mine data using an analytical engine
  - Supply detection queries for commonly known sequences – e.g. OWASP Top 10
  - Enable the addition of new query templates that handle common and proprietary sequences
  - Enable conversion of discovered stochastic sequences to detection queries