True Source Code Analysis for Security

October 20

2009

Maty SIMAN, CISSP

Contents

Abstract ... 3

Introduction to Compilation and Linkage ... 4

Benefits of True Source Code Analysis ... 5

- Introduction ... 5
- Code Fragments ... 5
- Non Compiling Code ... 6
- Cloud Compiled Language ... 6
- Non Linking Code ... 6
- Compiler Agnostic ... 7
- Platform agnostic ... 8
- Proprietary Resolving ... 9
  - Advanced Data members Distinction ... 9
  - Advanced Array Elements Distinction ... 10
  - Compile Time Resolving ... 11
- Compiler Optimization Compensation ... 12

Abstract

CISOs have responded to the sharp rise in hacking by asking developers and auditors to implement secure software development for in-house and outsourced code. In recent years, “source” code analysis has become the *de facto* choice to introduce secure development as well as gauge inherent software risk.

The irony is that source code analysis doesn’t often look at the source at all. In fact, the majority of the products are using Binary analysis or byte-code analysis (BCA) created by the compiler. This method saves a great deal of effort when developing the analysis tools, but lowers drastically the usability and accuracy of the results. For example, current technical approaches examine code so late in the development cycle or—worse—after development leaving a high volume of vulnerabilities undiscovered. For the unfortunate developer and auditor, they are technically incapable of delivering the CISO’s vision of secure software.

The differences between binary analysis and byte-code analysis have received little attention. This topic was addressed in just two recent blog posts [1, 2]. Worse, *true source code analysis* (TSCA) – which seems most logical for SCA, has been largely ignored. Yet only TSCA can deliver upon the CISO’s promise of building security in.

Further, with the onset of cloud computing there is a new breed of languages used mainly in cloud computing where the developer develops the code while the cloud platform provider is responsible for validation, proprietary compilation and execution of the programs. The code has no manifestation as byte-code nor as binary, and the SCA must be done on the source code itself. No static analyzer is properly equipped to address this growing, important segment.

This technical paper fills this gap and explains how developers, auditors and cloud platform providers benefit from deploying a *true source code analysis* tool. with detailed code examples.

---

[1] http://blogs.gartner.com/neil_macdonald/2009/07/24/byte-code-analysis-is-not-the-same-as-binary-analysis/

[2] http://www.veracode.com/blog/2009/07/bytecode-analysis-is-not-the-same-as-binary-analysis/

Introduction to Compilation and Linkage

Source Code Analysis is the technique of analyzing source code in order to retrieve valuable information about the application without ever executing it[3].

For purpose of illustration a well-known code-analysis tool is in fact the compilers' frontend. A compiler is the tool that transforms source code into machine code (or byte-code for managed languages). We can roughly divide the compiler into two parts, the *frontend* checks the syntactic and semantic correctness of the code and then loads a representation of source into memory. The second component, the *backend*, is used to write the representation back to disk in its machine-code form (*Object Files)*.

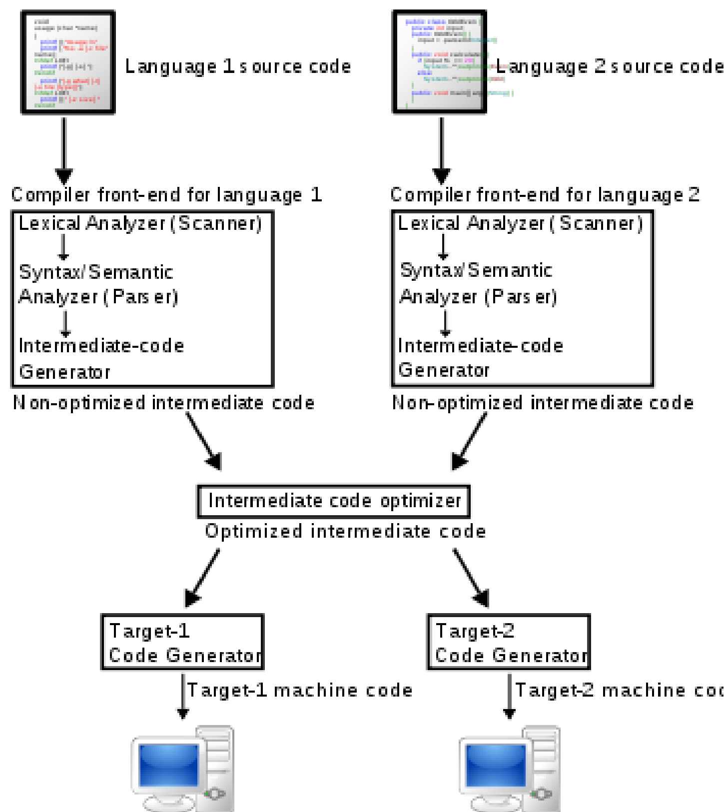


Figure 1 - An operation diagram of a typical multi-language, multi-target compiler.[4]

To complete the overview of the compilation process, we should also mention the linking phase. Most applications are comprised of several modules. Each resides in its own source file, which in turn, is transformed into object files, one for each module. A *linker* is a tool that takes the object files and combines them together and generates a single

---

[3] The technique of actually executing application in order to get data about applications' functionality is called *Dynamic Analysis* and is not covered within this paper.

[4] Wikipedia: http://en.wikipedia.org/wiki/File:Compiler.svg

executable file. While doing so, it *resolves* the types in the code – finds the correct definition and usage of each element in the code (variable, function, class, etc…). Importantly, the compiler's primary goal is to enable the optimized execution of the program in its target operating system and hardware. Unfortunately, this means that the ability of a compiler-based SCA tool to use compiler output in order to fully comprehend the code itself becomes seriously limited.

Benefits of True Source Code Analysis

Introduction

Take a good look at the following C#/.NET code example:

```
public void noSC()
{
    s = Request["Textbox1"];
    Response.Write ("See text below: ")
    Response.Write ("Is it vulnerable to XSS? ");
    Response.Write(s);
}
```

Is it potentially vulnerable to Cross-Site Scripting? Obviously it is.

However, none of the existing BCA tools will identify it. Only TSCA is capable of finding it! Why? A semi-colon is missing at the end of the second line and will fail any compilation process – necessary by BCA tools. The TSCA tool can "forgive" syntactical errors and still detect the problem.

The compiler performs many actions during its process in order to create the binary files. These actions were not designed for performing security analysis but rather to make the code as complete and efficient as possible - making the compiler way too restrictive for early security analysis in turn preventing the successful discovery of vulnerabilities.

This section will demonstrate some test cases where TSCA inherently performs better than BCA.

Code Fragments

In the code above there were not one but two errors that will prevent BCA from finding the vulnerabilities – the missing semicolon discussed before, and also the 's' variable that wasn't properly declared. A TSCA tool is capable is scanning code fragments.

Non Compiling Code

The code above, demonstrates the ability of TSCA to scan code not compiled due to syntactic errors. This capability allows developers to scan incomplete code, allowing the discovery of vulnerabilities much earlier during the Software development Life Cycle (SDLC).
This is not just a sales-pitch – actual vulnerabilities slipped through scanning in real-world projects, like the Linux OS (Finding Linux Bugs Before they Become Exploits[5]).

This example can be extended to non-compiling blocks, non-compiling functions (as seen above), non compiling classes, and non-compiling namespaces.

All these capabilities are supported easily and natively using TSCA.

Cloud Compiled Language

There is a new breed of languages used mainly in cloud computing where the developer develops the code while the cloud platform provider is responsible for validation, proprietary compilation and execution of the programs. The code has no manifestation as byte-code nor as binary and the SCA must be done on the source code itself.

The most known example is the Force.com platform supplied by Salesforce.com based on Apex as the server based language and Visualforce as the client based. Obviously, only TSCA product can support this new paradigm.

Non Linking Code

```
string s = Request["Textbox1"];
string s2 = "Select * from t_users where name = '"+s+"'";
Results = ExecuteSql(s2);
```

Is the code above vulnerable to SQL Injection?

Code auditors are familiar with the following scenario:

A customer sends you a source code of his application for your review. You look at the code, and see that it references many infrastructure libraries, which you didn't get their source. You load your favorite BCA tool, try to scan the code, but it doesn't work. It fails on “Missing Library” – and then you spend days building stubs for the missing parts just

[5] http://www.internetnews.com/dev-news/article.php/3831716

to make it work. A lot of hard work without any added value.

While a BCA product will fail to even start the scanning process, a TSCA product will easily identify the SQL Injection above, even if the actual code of ExecuteSql is missing. You were able to easily find it manually – why your tool can't do the same? TSCA can!

Compiler Agnostic

Compilers transform source code into binary/byte code. However, each compiler does so differently – and the output on the same source code varies from compiler to compiler. BCA are always on the endless race of supporting more and more compilers, since they have to read, understand and analyze the different outputs of different compilers. However, all of them must be able to read the same source – which must comply with a single standard. By contrast, a TSCA solution doesn't need to be part of that rat race – it only needs to understand the single standard of each language.

Following is a list that appears on the technical spec page of BCA provider:

- GNU GCC/G++
- ARM
- Microsoft Visual C++
- Green Hills
- Wind River Diab
- Sun Studio C/C++
- Freescale
- Metaware
- Hitachi h38
- IBM VisualAge C/C++
- Intel C++
- QNX Compiler

The longer the list is, we must admire their efforts in supporting all the platforms, but wouldn't it be simpler to just support the published C++ standard (ISO/IEC 14882:2003[6])? No need for upgrades / updates to support different compilers.

[6] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38110

Platform agnostic

We can take this idea one step further and come to the conclusion that there is no need for a compiler to actually be part of the code analysis process in order to find security vulnerabilities. This means that no matter what platform and OS are used to develop the code, the exact same TSCA can be used to scan any code anywhere – independent of the environment used for actual development. There is no need to have different scanning tools on different machines, only to support different development and execution platforms.
One of the BCA vendors is "proud" of supporting:
PLATFORMS: Windows, Solaris, Linux, Mac OS X, HP-UX, AIX

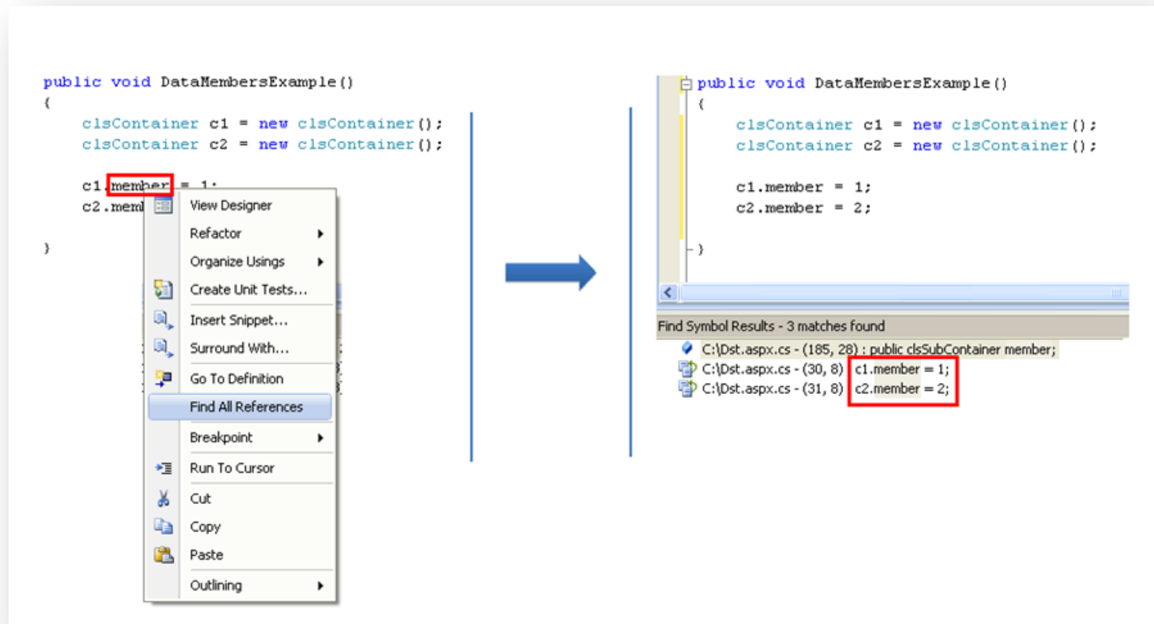
With TSCA, the above list is irrelevant – all platforms are supported immediately, no matter where the actual execution takes place.

Proprietary Resolving

Resolving is the process where the compiler/linker correlates between references of same code elements in different places in the application. The standard compilers/linkers perform the resolving process in a way that fits their own needs, but very often it is not aligned with the needs of finding security vulnerabilities. This section describes the tweaks that can be done at the resolving stage of TSCA in order to improve the process of finding security vulnerabilities.

Advanced Data members Distinction

By default, compilers treat similar data member of different object as the same reference. This can be seen for example by using Visual-Studio's "Find All References" function:



It can be easily seen that VS.Net returned both c1.member and c2.member, although they are completely two different objects. This may lead to false positives and false negatives during the process of finding security vulnerabilities.

```
    clsContainer c1 = new clsContainer();
    clsContainer c2 = new clsContainer();
    c1.s = Request["Textbox1"];
    Response.Write(c2.s);
```

So, is there an XSS here? No but many BCA tools falsely identify this issue as XSS, leading to a false positive.

Advanced Array Elements Distinction

Advanced compilers treat all array elements as the same object (i.e. Viewstate[“a”] and Viewstate[“b”] are considered as the same object; although, obviously they are not)

```
Viewstate["key1"] = Request["Textbox1"];
Response.Write(Viewstate["key1"]);
Response.Write(Viewstate["key2"]);
```

The code above contains a single XSS. Most BCA will either find two XSS (one of which is False Positive), or won’t find any (meaning one False Negative). By using advance array elements distinction technique at the proprietary resolving stage, TSCAs are able to find the one XSS in that code.

Compile Time Resolving

Some types of code structures are traditionally resolved at runtime and not during compilation as more information is needed. An interesting point is that even if the compiler DOES have the necessary information, it will still wait with the resolving until the application is actually being executed. Let's look at polymorphism as an example of this issue. Examine the following code:

```
clsContainer c1;
clsSpecificContainer c3;

. . .

c1.s = Request["Textbox1"];
c3.s = Request["Textbox1"];

. . .
c1.printSelf();
c3.printSelf();
```

Assuming that "clsSpecificContatiner" inherits from "clsContainer", we don't know what instance of "printSelf" will be executed at the line "c1.printSelf()" as it depends on the actual object assigned into c1 (either clsContainer.printSelf() or clsSpecificContainer.printSelf()); However, we do know that clsContainer.printSelf won't be executed at line "c3.printSelf". Still, the compiler doesn't make use of this extra information, and leaves both lines as virtual-call to clsContainer.printSelf:

```
    L_0055: ldloc.0
    L_0056: callvirt instance void clsContainer::printSelf()
    L_005b: nop
    L_005c: ldloc.1
    L_005d: callvirt instance void clsContainer::printSelf()
    L_0062: nop
```

Wouldn't you expect to find a virtual call to clsSepcificContainer ::printSelf on line L_005d.

This has great impact on the analysis performed by BCA – as they have difficulties in finding the appropriate call, since the compiler "masks" the relevant information. TSCA is capable of easily narrowing down the results to the relevant calls only.

Compiler Optimization Compensation

One of the many roles compilers fulfill is to optimize the created code in terms of efficiency and size. For example, compiler might remove “irrelevant” line, or dead blocks. Developers often ask to see vulnerabilities within dead blocks – for example, as part of their debug process:

```
public void DeadCode()
{
    bool debug = true;
    string s = Request["Textbox1"];

    if (debug == true)
    {
        Response.Write("Data retrieved");
    }
    else
    {
        Response.Write("Hello " + s);
    }
}
```

Tools that rely on compiled binaries fail to address such request. They are completely blind to the content of “dead code” and miss the XSS on the code. Although this XSS manifests itself only on production, it’s extremely important to know about its existence during development.

With TSCA, the developers can choose whether to see vulnerabilities in dead code or not, which are left out of reach from BCA tools.