The Need for Fourth Generation Static Analysis Tools for Security – From Bugs to Flaws

By Evgeny Lebanidze
Senior Security Consultant
Cigital, Inc.

This paper discusses some of the limitations of the current (third) generation static code analyzers for security available on the market today and gives reasons for the plateau in their usefulness to a code reviewer. We further describe some of the characteristics of the next generation static analysis technology that will enable a new quantum leap in the space of static analysis with tools that are able to detect software security flaws, not merely implementation level bugs.

Introduction

There have undeniably been some significant advances in the state of the art of static analysis for security since the beginning of the millennium. As with almost any type of tool, these tools yield the most value in the hands of an experienced operator (e.g. expert code reviewer). However, in our experience the usefulness of the current generation static analysis tools is limited in that they are not likely to find more than 15% of the software security weaknesses that are ultimately discovered during the code review. Furthermore, experience shows that the most significant software weaknesses are rarely found by static analysis tools directly, even though in some cases sifting through the result set produced by these tools may point the reviewer in the right direction and focus their attention on the vulnerable piece of code, thus facilitating discovery of the big problems.

To understand why a code reviewer cannot presently derive more utility from static analysis tools one needs to understand the difference between implementation level bugs that have adverse security implications and software security flaws.

As an example, failure to ensure that the destination buffer is large enough to accommodate the contents of the source buffer when using a C function *strcpy* is an implementation level bug that may lead to the infamous buffer overflow problem. On the other hand, using the current system time in milliseconds as a way to generate "random" session ids for web application users is a software security flaw that may enable a session hijacking attack. As a rule of thumb: implementation bugs are almost always tied to a particular programming language, while software flaws tend to be more implementation agnostic. Static analysis tools of today have a chance of detecting bugs, but not flaws.

The remainder of this paper defines the generations of static analysis tools and provides some insight into the value the current static analysis tools offer in practice to software security consultants today. We further describe the plateau in third generation static analysis technology and provide some characteristics of the next (fourth) generation static analysis tools that will help an expert code

reviewer detect more software security problems (flaws).

First Generation Tools

First generation of static analysis tools was mostly homegrown scripts that did little but apply 'grep' like techniques directly to the source code in order to detect usage of potentially unsafe functions. These tools would for instance look for usage of functions such as *gets* in C and report that a buffer overflow is possible. Most of these tools would not be able to tell if *gets* was a C function, a variable "boolean getsentence" or part of a comment such as "A gets the value of B". These early tools would essentially perform a keyword search and report on potential problems typically associated with the functions being used. While a small improvement over the status quo, this shallow analysis did not provide much real value and resulted in a tremendous amount of false positives for the analyst to sift through.

Second Generation Tools

Second generation static analysis tools were essentially open source tools developed by software security industry gurus and released to the public. This generation gave birth to tools like RATS from Secure Software, Flawfinder developed by David Wheeler and ITS4 from Cigital. There are several improvements that were made in this generation of static analysis.

1. The analysis would no longer be performed on source code directly, but rather on the tokenized version of the code resulting from lexical analysis. That means that the analysis tool could now differentiate between *gets* being a function name and the word "gets" used in a comment. "grep" like techniques were now applied to program representation resulting from lexical parsing.

2. The vulnerability knowledgebase became separated from the engine code and pulled out into a separate repository (e.g. XML file). This made the vulnerability knowledgebase easier to extend.

3. The vulnerability knowledgebase significantly grew in size, covering more than just the most frequently used dangerous functions. Also, there was an attempt to cover more types of software security weaknesses, such as Time of Check Time of Use (TOCTOU) problems.

4. Some tools also expanded to cover more than just C/C++ languages (e.g. RATS also covers Perl, PHP and Python).

Software developers and security consultants alike initially got excited about these new tools, but then quickly realized their limitations and the number of false positives that they had to go through. While these tools remained in limited use, most of the community became disillusioned with static analysis for security and wrote it off as a problem that would be too hard to solve. It became clear that a fundamentally new approach would be needed if static analyzers were to be useful to the industry.

Third Generation Tools

Third generation of static analysis tools for security was driven by commercial vendors backed up by venture capital firms who began to see an emerging market for these kinds of tools. This resurgence was also fueled by significant advances in computer hardware, specifically CPU speed and larger RAM.

Interesting things began to happen around 2002-2004 with several companies becoming players in this field including Fortify Software, Secure Software, Ounce Labs and Coverity.

The main reason for the sudden commercial push was that the extent of the software security problem and the real business impact of vulnerable software finally began to sink in. Additionally, advances in hardware offered hope that approaches deemed previously computationally impractical could now be within reach.

The industry also began to understand that the problem will not be solved by perimeter defenses (e.g. firewalls, IDS, etc.) and really had to be addressed at the root, the software layer. Additionally, unlike dynamic analysis tools and penetration testing (i.e. black box testing), static analysis tools at least in theory offered the promise of quickly dissecting large quantities of code in an automated fashion, pinpointing the exact location of the security bug, thus reducing the overall discovery/remediation cycle.

It is important to note that software vendors building third generation static analysis tools took substantially different approaches. Many concepts from classical computer science from the field of compiler theory and algorithms have been combined with techniques such as taint propagation (to support data flow analysis) to develop the juice behind the analysis engine. Some of these techniques were proprietary and the approach taken reflected the philosophy of the chief scientist: choosing depth vs. breadth or increased accuracy vs. time to complete the analysis.

In this paper we do not attempt to describe the approaches taken by each of the vendors, but rather generalize a few of the commonalities for the purpose of discussion. Third generation of static analysis tools largely had the following characteristics:

1. Analysis was performed on program representation after the parsing stage, thus performing the analysis on the abstract syntax tree (AST) of the program. Some vendors went a step further generating their own internal representation from the AST on which to perform the analysis. With this approach, the software needs to compile in order to be fully analyzed.

2. The analysis was no longer limited to "grep" like techniques, although that approach remained partially in use to find some of the really low hanging fruit issues. Mostly this generation of static analysis is famous for introduction of control flow (intra and sometimes even inter procedural) and data flow (using taint propagation) analyses, as well as analyses leveraging understanding of syntactic and semantic structure of the application.

3. Some tools also employed the concept of modeling. The analysis engine would accrue certain information and understanding of the program and update the program model in the process. Rules in the knowledgebase could then leverage the information in the program model to achieve higher accuracy.

4. Some vendors took the modeling concept a step further to attempt to simulate execution (symbolically) of the code being analyzed, keeping track of conditional states of the program and merging these states when possible. In some sense that approach would treat the program itself as byte

code running inside a virtual machine (the analysis engine) and the vulnerability knowledgebase would have the ability to query the state of the program during its simulated execution to gain the information that it needed to determine whether a weakness exists. This approach required a version of the code base that could be compiled and also required more time to conduct the analysis, but at least in theory allowed for higher accuracy findings.

5. The tools also included functionality necessary for commercial users, such as advanced filtering/suppression capabilities, reporting capabilities, false positive management capabilities, extensibility of rules, integration with build environments, etc.

6. Tools began to cover more languages and supporting frameworks. For instance, they would cover not only Java, but also J2EE and various commonly used open source frameworks (e.g. Struts, Spring and Hibernate).

7. There was greater flexibility and power in knowledgebase customization to support creation of custom rules to check for best practices using custom frameworks, enforcement of coding standards and generally to help the tool understand custom code.

There are more identifying characteristics for the third generation static analysis tools, but the ones mentioned above are the main ones. With this generation the tools have clearly come of age and began to get slowly adopted by the industry. Early adopters were followed by mainstream customers (e.g. Oracle) purchasing static analysis technology. Companies began to roll out these tools to their development organizations and security consultants started using these tools extensively in their practice.

Static Analysis Tools in Practice

Despite the message currently emanating from many vendors of static analysis tools that their products if consistently used by the development organization will ensure the production of secure software, people without vested interest in selling these tools who are familiar with the capability of static analysis tools and also understand software security will likely agree that these tools are very far from being a magic wand solution.

The old saying goes that a "fool with a tool is still a fool", but that is not even the whole story. While it is true that an expert security code reviewer will get a lot more value out of static analysis tool than a novice and will be able to sift through the result set and validate the findings much more effectively, even he or she will quickly reach the limit in the benefit that can be derived from the tool. In our personal experience as security consultants who constantly review software for security, it is rare for a tool out of the box (without customization) to find more than 15% of the ultimate findings of the assessment, and these will rarely be the most serious security issues found. That 15% seems to be a plateau in the usefulness of the static analysis tools that has not changed for at least two years. Put quite simply, if a security review of software relies solely on the results of a static analysis tool, that review is very far from being sufficient and it is very likely that only the low hanging fruit type of security issues have been found. Thus there is no substitute for an expert human security auditor who will use the tools, but will also conduct an extensive manual

code and design review leveraging his or her experience.

Furthermore, just like a carpenter will likely have more than one type of hammer, the security reviewer should use more than one type of tool during the review and pick the best one for the job. Each of the commercially available tools have their own strengths and weaknesses, and often it is hard to predict a priori what tool will provide the most value on a particular code base without actually performing the analysis. After an initial analysis it might become apparent what is the right tool or the right combination of tools for the code base.

Reasons for the Plateau

The reader may now wonder why this plateau in the usefulness of third generation static analysis tools exists despite the significant improvements in the way the analysis is performed, advances in hardware and the growth of the vulnerability knowledgebase. We have already hinted at the reason: fewer security weaknesses in modern applications are due to implementation level bugs and more are attributed to software security flaws. Static analysis tools today cannot catch software security flaws. In most cases, even most of the implementation level bugs go undetected.

For instance, how would a static analysis tool detect that the authorization service of the application can be bypassed? How would it understand that access to a particular functionality of the application needs to be preceded by an authorization check that the programmer forgot to add? Or how would a static analysis tool realize that the application does not enforce a sufficiently strong password policy or even that usage of SSL enabled sockets is broken? These are the types of security flaws that often have the most serious implications and static analysis tools today have no hope of finding them.

In software written today, more and more of the serious security weaknesses tend to be flaws and not bugs. One of the main reasons for this phenomenon is that programmers use languages where they are expected to take care of a lot of low level implementation constructs and details such as memory management less frequently (e.g. C/C++) and instead prefer managed languages like Java and C# that hide most of the low level detail from them.

On the other hand, software is becoming more conceptually complex with unprecedented integration and interoperability requirements. Consequently, more complicated architectures and designs often introduce serious security flaws into an application before a single line of code is even written. Furthermore, understanding of the many APIs made available by the modern programming languages and supporting frameworks and how they should be used in a secure fashion can be a daunting task for a software developer. In a way the software security problem has shifted to a higher plane from implementation level bugs to software security flaws that are found both at the architecture/design level and at the implementation (mainly API) level. Thus the static analysis tools will need to have a greater ability to detect flaws if they are to be more useful to a code reviewer. These tools will never replace an expert software security auditor, but they have a chance of bringing more value. That however will not happen until the quantum leap to the fourth generation of static analysis.

This is a gap that cannot be traversed with the current generation of static analysis

technology. Fundamentally more advanced analysis will be required if these tools are to become more useful to the security code reviewer and allow for the plateau to be overcome.

Tools of the Next Generation

The big question to answer is what kinds of improvements need to be made in the state of the art of static analysis for security to enable the tools to catch more software security flaws? What characteristics will these next (fourth) generation static analysis tools have?

Next generation static analysis tools for security will probably require a closer relationship between the tool itself and the expert code reviewer. Meaning that the code reviewer will need to have a way to feed certain information into the tool and the tool will then use that information to perform more accurate high level analysis that can in fact detect some of the security flaws. For instance, a code reviewer might tell the tool that a particular method represents an entry point for performing authentication in the system. The code reviewer might also tell the tool what argument represents the user name and what argument represents the password used as log in credentials. The tool might then have a chance of figuring out that sufficient password complexity is not being enforced by the system or perhaps even determining ways in which authentication can be bypassed.

A reviewer should also have the ability to set "breakpoints" in the analysis execution in order to do things like set parameter values and specify control flow in the analysis engine. The latter will help the tool deal with dynamic invocations such as in cases when reflection is used.

The tools of the next generation will need to have more understanding about the semantics of the code (i.e. what it actually does) and not just the syntax. Some of the semantic information can be derived through a probabilistic process (and perhaps confirmed by the code reviewer) and other semantic information will have to be directly entered by the code reviewer into the tool. Use of code annotations during security analysis may be the first step in this direction.

Fourth generation static analysis tools will also need to do more modeling to understand the interactions between the various pieces of the program and their effect on program data and program states. This enhanced modeling ability and a more holistic understanding of the application will increase the accuracy of the tools with detection of the bugs that they can find today and will also be essential for detection of the security flaws. In a true sense, static analysis tools will have to become more than just that. They will have to more effectively simulate the execution of the program and consider possible states that might result at runtime.

The tools of the next generation will also need to have a deeper understanding of the various third party APIs used by the application and whether they are used securely. Many implementation level flaws result from misuse of these third party APIs and the tool needs to understand the possible issues there.

Next generation tools might also have the ability to take in some design level information about the system in some standardized form and then use it to determine whether the security mitigations mandated by the design have been properly implemented. This problem, if generalized is probably very hard (if not

impossible) to solve, but a solution might be achievable for some specialized cases.

There are likely to be other significant advances in next generation static analysis technology. Just like advances in the current generation to introduce control and data flow analysis have borrowed heavily from compiler theory and algorithms, analysis engines of the next generation might borrow from other areas of computer science such as predictive modeling, advanced algorithm theory and even artificial intelligence. Continued advances in hardware will no doubt help.

Conclusion

While nobody has a crystal ball to see the future, one thing is relatively clear: A quantum leap in the way that security static analysis is performed will be needed if code reviewers are to gain substantially more utility out of the tools and if the plateau is to be crossed. While tools are very unlikely to ever replace an expert human auditor, if they could detect some security flaws and not merely bugs their value would be significantly higher.

Many ideas for improvement are likely to come from universities and other research organizations so tool vendors should follow developments in the academia closely. In the meantime, if someone tells you that all you need is a tool to find all of the security problems in your software, you will know better.

Disclaimer: The views expressed in this paper are these of the author and do not necessarily reflect the official views of Cigital, Inc.

This paper is licensed under the Creative Commons Attribution-ShareAlike 2.5 License.