Cyber Security at Software Development Time

Mark Bradley, Ansgar Fehnker, Ralf Huuck

National ICT Australia (NICTA)
University of New South Wales
Locked Bag 6016
Sydney NSW 1466, Australia

***Abstract*—Secure systems are intrinsically dependent on secure software. Creating secure software is no simple task and every aspect of the software development lifecycle has to be taken into account. In this article we focus on security in the software implementation phase and present a number of techniques that enable the formal checking of security properties at software development time. We give an overview of some of the automated analysis techniques available today ranging from tree-based pattern matching to model checking. Moreover, we present our source code analysis tool Goanna which integrates those security analysis techniques, and we provide a number of application examples, where Goanna detects real security threats demonstrated in application examples from the National Institute of Standard's comparative exposition.**

***Index Terms*—Security, Static Analysis, Model Checking, C/C++, NIST, Tools**

I. Introduction

Critical infrastructure such as energy, transportation as well as their encompassing businesses are increasingly controlled by software [1]. In fact, software is at the heart of many mission and safety critical operations and the correct behavior of this software is essential. Unfortunately, the development of software is progressing at a much faster pace than the careful design and (formal) verification of safety and security properties within software. Hence, new methods are required that ensure an acceptable level of software security right from the onset.

Cyber security needs to be included in all aspects of the software development lifecycle, not only in the system architecture, the design, and testing, but also in the software development process and the code implementation. One of the reasons is that implementations which are not written carefully with security aspects in mind, open up potential attack vectors that can be exploited by malicious software or hackers. Examples of programming flaws that compromise the security include SQL injections, buffer overruns and memory corruptions.

While software is developing at such a rapid speed and is constantly growing in complexity, traditional software assurance methods such as code review, (penetration) testing and auditing do not scale in the same fashion. The need for automated tools is high.

In this work we explain a number of automated techniques for detecting security issues in source code. Moreover, we present their implementation in our source code analysis tool *Goanna*. Goanna belongs to the class of *static analysis* tools and has been used to identify many real-life security threats in the past. In this work, we highlight the underlying architecture of Goanna and present a number of case studies taken for the recent National Institute of Standard (NIST) and US Department of Homeland's static analysis tool exposition. All of the examples in this work are from well known existing code bases, demonstrating real security issues at source code level.

This work is organized as follows: We will give an introduction to automated static code analysis techniques for security in Section II. The integration of these techniques in our source code analysis tool Goanna and its underlying architecture is described in Section III. Afterwards, in Section IV, we present a number of real-life security threats that have been automatically identified by our tool and confirmed by NIST. The results of that section are the basis for the following discussion and future outlook in Section V.

II. Static Analysis for Source Code Security

The term *static code analysis* comprises of a number of automated techniques that analyze code without executing it. The goal is to understand the source code as much as possible and in turn detect security and mission-critical safety issues. An example of static analysis would be a procedure that first determines where memory is allocated and de-allocated, and then examines if there is a program path that does not clean up the memory (de-allocating it). This path would lead not only to a memory leak, but also to a potential information leak.

Static code analysis has been the subject of much research in the past [2], [3]. It is, however, only in the recent years that it moved from compiler and source code optimization to security and quality analysis [4]. Some of the recent advances include the use of new techniques for an efficient whole-program analysis and for an exploration of most, if not all program paths, while being fast and keeping the number of false alarms low.

In the following we describe the main techniques used by modern static analysis tools for detecting security vulnerabilities in source code.

A. Tree-Pattern Matching

Pattern matching is the simplest technology applied by virtually all static analysis tools. Pattern matching means to identify points of interest in the program's syntax, typically by defining queries on either the plain source code or on the *abstract syntax tree* (AST), i.e. the representation of the source code after parsing. For instance, it is well known that in C/C++ programs the library function `strcopy` is vulnerable to security exploitation if not used absolutely correctly. Hence, a simple keyword scan for `strcopy` can help with this. Queries can become more complicated, and a series of interdependent queries may be used for more advanced checks, for example to identify inconsistent use of semantic attributes. Our analysis tool Goanna uses such tree-pattern matching [5] on the AST. However, pattern matching is fundamentally limiting, since it only searches for keywords and their context, but is unable to take control flow, data flow, or other semantic information into account.

B. Data Flow Analysis

The next step up in terms of technology is about understanding how certain constructs are related. Data flow analysis [2] makes use of the structure of program, in particular its *control flow graph* (CFG). It is a standard compiler technique to examine the flow of information between variables and other elements of interest that can be syntactically identified. An example is checking for uninitialized variables leading to unexpected arbitrary behavior. We can syntactically identify program locations that are variable declarations and uses of variables either as a reading operation (such as the right hand side of an assignment) or a write operation (such as the left hand side of an assignment). Data flow analysis enables us to examine if there is a read operation to a variable without a prior write operation. These and similar methods are used to analyze, how certain elements in a program are related. Typically, however, data flow analysis uses a number of approximation techniques that prohibit the precise identification of the program pat. To remedy this *model checking* can be applied.

C. Model Checking

Model checking was developed in the early 1980s as a technique to check whether larger concurrent systems satisfy given temporal properties [6], [7], [8]. It is essentially a technique to determine whether all paths in a graph satisfy certain ordering of events along that path. Unlike other techniques that enumerate paths model checking does not put a limit of the number, the length or the branching of paths. Another advantage is that if it finds a violation, it will also return a counterexample path. For their solution the original authors received the Turing Award in 2007.

Model checking was originally applied to the formal verification of protocols and hardware designs. In recent years, a strong push has been made towards software verification, and effective methods have been developed to overcome scalability challenges. Or tool Goanna is the first tool that manages to apply model checking on a grand scale to static program analysis. This means that we use it to analyze millions of lines of code for over one hundred different classes of checks. The advantage of this approach that is oftentimes faster than existing data flow and path enumeration approaches. Moreover, it also tends to be more powerful as it allows specifying complex relations between program constructs.

D. Abstract Data Tracking

To detect buffer overflows, division-by-zero errors and other defects it is helpful to know as much as possible about the values that can occur at certain program locations. A technique to approximate and track data values is commonly called abstract interpretation [9]. Abstract interpretation estimates for every variable at every program location all the potential range of variables. For example, for an array access, all potential index values can be estimated. Different domains can be used for data tracking, with varying levels of precision. There is a trade-off between the precision and speed of the analysis. Goanna uses a variant of interval constraint analysis that is reasonably precise while fast, but can also use automated theorem proving techniques (SMT solving) to validate individual program paths.

E. Whole Program Analysis

While the aforementioned techniques help to detect various security issues and are often complimentary, one of the key factors to success is to scale to large code bases. Many potential issues require to understand the overall call structure of a program, and being able to track data and control flow across function boundaries. To overcome this challenges advanced tools introduce the ability to generate *function summaries* automatically. These summaries contain only the essential information of a function that is needed for particular security check. Instead of propagating information of the whole function, which can be prohibitively large, only the summary information is used. Generating summaries is an iterative process taking the mutual impact (from recursion etc.) into account.

III. The Goanna Static Analysis Tool

Goanna is an automated analysis tool that does not execute code as in traditional testing, but examines the code structure, the keywords, their inter-relation as well as the data and information flow across the whole source code base. These techniques can be fully automated and scale to millions of lines of code. As such Goanna is able to identify many classes of security issues automatically at software development time. Moreover, Goanna is the only tool that combines all the techniques mentioned in Section II. The Goanna tool is fully path-sensitive and performs whole-program analysis. For a detailed technical description of the underlying formal techniques see [10].

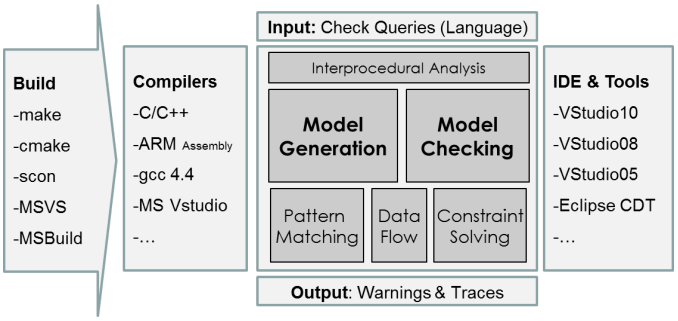

Fig. 1. Goanna static analysis architecture

A. Supported Languages and Architectures

The Goanna tool is currently implemented to handle C/C++ source code. The architecture and the technology are in theory adaptable to other imperative programming languages such as $C\#$ or Java. However, the C/C++ programming language is still predominant in many mission-critical systems while at the same time it easily suffers from potential exploits.

The Goanna tool fully parses C/C++ source code including compiler specific dialects and extensions such as GNU, Visual C++ or GreenHills. It is available for Microsoft Windows as well as various Linux on either 32-bit or 64-bit machines. On top of that Goanna supports the cross-compilation to a number of embedded platforms. It is possible to integrate the tool in popular build environments, such as *make* or *scons*.

B. Analysis Framework

Figure 1 depicts how the different types of analysis are embedded in Goanna. The core of the tool uses model checking, while the other techniques are used for particular checks, and to assist in the generation of the models. A distinguishing feature of Goanna is that it architecturally modularizes the core analysis engine and the checks that are performed. Goanna has a specification language to define checks; this allow for the rapid development of new checks, separately from the core algorithms, and any future improvement of the engine will be effective for all checks.

In short, Goanna uses model checking for all path-sensitive check on how program information relates. It uses tree-pattern matching to identify certain locations and operations of interest and abstract data tracking by constraint solving for specialized checks, such as buffer overruns, shifting beyond bounds and overflow errors. Moreover, Goanna uses a number of heuristics and advanced features such as interprocedural whole-program analysis to achieve speed and scalability.

C. Reporting and Development Environments

As mentioned earlier, all analysis runs by Goanna are fully automatic and do not require human interaction. The analysis results can be viewed in a number of ways: Either as a simpler compiler-style output on the command line, as a pre-defined XML report or as an integrated web-page with in supported development environments. Currently, these environments are the Eclipse IDE and Microsoft Visual Studio.

Moreover, it is possible to configure the Goanna for project specific needs either through one of the IDEs or through text-based configuration files that can be shared among developers.

IV. Detection of Real-life Security Threats

This section describes a number of confirmed security issues that have been successfully detected by the Goanna tool. All of the examples are part of the 2010 *Static Analysis Tool Exposition (SATE)* [11]. This exposition is run by the National Institute of Standards (NIST) and the Cyber Security Devision of the Department of Homeland Security.

The case studies comprised the open source secure IMAP and POP3 server Dovecot, the network protocol analyzer Wireshark and the Chrome operating system. The size of the code bases in SATE differed greatly. Dovecot is 360 kLoC in size, Wireshark 1.2.0 and 1.2.9 are both 1.7 MLoC, and Chrome 5.0.375.54 and 5.0.375.70 are 1.5MLoC and 1.7MLoC, respectively. The number of warnings Goanna issued range from 180 for Dovecot, 534 and 532 for the Wireshark code bases, to 1079 and 1173 for the Chrome code bases, respectively.

In the following we describe number of representative security issues and highlight the type of analysis technology that would generally be required to detect them.

A. Failed Error Handling Routine

The following example, taken from Dovecot, highlights a failure in proper error handling leading to an unexpected program behavior that is potentially exploitable. To detect this issue whole-program analysis and abstract data tracking was required.

```
director-connection.c:655: warning: Goanna[RED-cmp-never] Comparison never holds
655 if (str_array_length(args) != 2 ||
        director_args_parse_ip_port(...) < 0) {
          i_error();
          return FALSE;
    }
```

The above code fragment is part of an error handler; the error routine `i_error` will be called if the output of `director_args_parse_ip_port` is negative. However, the output range of this function is in the interval $[0, 1]$ and will thus never be negative. Closer inspection showed that this code was refactored to return 0 if an error is detected rather than -1. Except for a few exceptions, the output of this function was treated like a Boolean. The mistake presumably entered since manual refactoring did not change all occurring tests in the error handler to use a Boolean condition. Abstract data tracking was used to determine the output range of `director_args_parse_ip_port`. This range was then part of the whole-program summary and subsequently used to detect unsatisfiable conditions in other functions.

B. Denial of Service

The following example, from the Wireshark code base, highlights a potential denial of service exploit. Goanna required its

model checking capabilities to detect this security issue. The surrounding code is:

```
packet-smb.c:8211: warning: Goanna[PTR-param-unchk-some] Parameter 'nti' is not checked against NULL before it is dereferenced on some paths, but on other paths it is
```

```
    case NT_TRANS_IOCTL: [...]
8211  dissect_smb2_ioctl_data(ioctl_tvb, pinfo,
          tree, top_tree, nti->ioctl_function, TRUE
          );
      [...]
    case NT_TRANS_SSD:
      if(nti){switch(nti->fid_type){ [...]
```

The detected issue resides in a longer switch statement. In one case `nti` is not checked to be not null before it is dereferenced, in the other case it is. This points to an inconsistency, which can lead to a null-pointer dereference. This particular example was reported as CVE-2010-2283 in the MITRE bug database.

Finding this bug requires to compare behavior along two paths. In this example, the inconsistency happens within a few lines. In general inconsistent paths may contain conditional jumps and loops, such that an explicit path enumeration becomes infeasible. Model checking provides algorithms to checks this exhaustively and efficiently.

C. Potential Program Crash

The final warning we like to discuss is also taken from the Dovecot code base. Goanna used model checking and interprocedural whole-program analysis to detect this defect.

```
director.c 180: warning: Goanna[PTR-null-assign-fun-pos] Dereference of 'preferred_host' which may be NULL
```

```
    *preferred_host =
        director_get_preferred_right_host(dir);
                  [...]
    if (cur_host != preferred_host)
180   (void)director_connect_host(dir,
          preferred_host);
    else {[...]}
```

The detected issues is based on the fact that the function `director_get_preferred_right_host` may return a NULL pointer and assigns it to `*preferred_host`. This is later used as parameter of `director_connect_host`. However, inter-procedural analysis shows that there exists a path in `director_connect_host` where this parameter will be dereferenced, without a prior check for being potentially NULL. The dereferencing of this NULL pointer can lead to an exception/crash that enables an attacker to potentially enter the system.

V. Conclusions

In this work we highlighted the importance of secure source code for developing secure systems. We presented a number of techniques that can help to improve cyber security right out the outset by analysis source automatically and efficiently. Moreover, we presented the integration of theses analysis techniques in our tool Goanna and illustrated the interrelation of the various components. Finally, we presented a number of real-security issues that have been detected by Goanna and explained how it requires the interaction of the previously introduced techniques.

We believe, that static program analysis in general is a valuable and cost-effective measure to cyber security. Valuable, because it enables the detection of many issues that are easily missed even by careful human inspection and cost effective, because all of the presented techniques can be executed fully automatically without human interaction. Moreover, applying these techniques at the coding stage, i.e., early in the software development lifecycle helps to reduce the overall software development costs for creating cyber secure systems.

For future work we anticipate new technologies and new standards to improve security further. Technologies such as SMT solving provide promising opportunities to deliver more precise results and to drill deeper in source code that before. New standards will help to increase the overall security awareness while helping developers and tool manufacturers to create and enforce higher quality software.

Acknowledgment

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] DHS Science and Technology, "Cyber security research and development broad agency annoucement (Baa 11-02)," Available at `https://baa2.st.dhs.gov`, January 2011.

[2] F. Nielson, H. R. Nielson, and C. L. Hankin, *Principles of Program Analysis*. Springer, 1999.

[3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[4] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," in *3rd International Workshop on Systems Software Verification (SSV 08)*. Submitted for publication, 2008.

[5] M. Benedikt, W. Fan, and G. M. Kuper, "Structural properties of xpath fragments," in *ICDT '03: Proceedings of the 9th International Conference on Database Theory*. London, UK: Springer, 2002, pp. 79–95.

[6] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons for branching time temporal logic," in *Logics of Programs Workshop*, ser. LNCS, vol. 131. Springer, 1982, pp. 52–71.

[7] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *Proc. Intl. Symposium on Programming, Turin, April 6–8, 1982*. Springer, 1982, pp. 337–350.

[8] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. [Online]. Available: http://portal.acm.org/citation.cfm?id=1382431.1382534

[9] P. Cousot, "Semantic foundations of program analysis," in *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981, ch. 10, pp. 303–342.

[10] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch, "Model checking software at compile time," in *Proc. TASE 2007*. IEEE Computer Society, 2007.

[11] M. Bradley, A. Fehnker, R. Huuck, and P. Steckler, "Goanna static analysis at the nist static analysis tool exposition," NIST, Tech. Rep., 2011, to be published.