Flawfinder and Secure Development Processes

Kevin Kredit *GVSU CIS Department*
Grand Rapids, Michigan
Email: k.kredit.us@ieee.org

***Abstract*—Static analysis tools provide insights to security and correctness early in the software development process. Flawfinder is a static analysis tool for C and C++ focusing on dangerous method use. It performs a syntax-aware search, identifies dangerous functions uses, and prints a ranked list of warnings. A complete secure development process may employ multiple static analysis tools. Statick is a plugin-based framework to manage multiple tools, including Flawfinder. To simplify the workflow for small projects, a Docker container has been built with a portable Statick environment.**

***Index Terms*—Flawfinder, Statick, static analysis, secure software, development process**

I. Introduction

Security cannot be bolted on. The sooner project developers consider security, the cheaper it is to achieve. If a project can continuously validate application security principles before it is finished, or even before it builds, it will experience fewer costly surprises later on.

Static analysis tools provide this continuous validation capability. They read source code and report possible errors in a developer-friendly format. Other validation techniques such as functional tests, unit tests, binary analysis, and runtime analysis require varying levels of product readiness. Because static analysis tools require only source code, developers can integrate them from the beginning of the process.

Flawfinder is a static analysis tool for C and C++ [1]. It works by searching for invocations of dangerous or hard to use functions, ranking them based in order of risk, and displaying them with helpful messages.

This paper gives a brief overview of static analysis tools in general, including their purpose, their capabilities and limitations, and how they fit in a secure software engineering workflow. It then describes Flawfinder in particular, including how it works, how it is configured, and example output from running it against a real project. Finally, it strives to lower the barrier for static analysis tool usage by describing the Statick framework and introducing a project that provides the Statick environment in a portable Docker container.

II. Static Analysis Tools

Static analysis tools read source code files and produce warnings related to incorrect or dangerous constructs. Because they do not compile or execute code, there are limitations on what types of errors they can detect. Therefore, they comprise only a part of a complete secure software development process.

A. Capabilities

Analyzing source code provides several capabilities.

a) Analyze during development: As already established, considering security late in the development process is difficult and costly. A major advantage of static analysis tools is that they do not require a complete product to test; they do not even require code to compile or contain valid syntax. Teams can setup static analysis tools at the start of a project and reap the benefits from the first line of code.

b) Correct little human errors: Humans make frequent trivial errors. By negligence or ignorance, developers make typos and violate coding standards. Working with editors and compilers, source code analysis tools can identify and correct spelling errors and enforce coding standard rules.

c) Identify dangerous method usage: Many standard software libraries contain dangerous or difficult to use methods. For example, the C standard library's `strcpy` is highly vulnerable to buffer overflows, and the safer `strncpy` should be used instead. `strcpy` cannot be removed from the library for the sake of backwards compatibility. To prevent its use in new code, static analysis tools can detect such usage.

d) Identify dangerous syntax idioms: Sometimes syntactic features that lead to concise code can also lead to confusion, and confusion leads to errors. Static analysis can identify these dangerous idioms.

e) Apply rigorous logical analysis: Advanced static analysis can detect errors that would take a human a long time to identify. Through control or data flow analysis, it can detect dead code, memory errors, and information leakage. This type of advanced analysis is typically computationally expensive and limited by the semantics of the language under analysis (e.g., the rules of Rust enable stronger static guarantees than C).

B. Limitations

Analyzing source alone also entails limitations.

a) No functional testing: Source code analysis is not intelligent enough to tell whether code is semantically correct. Functional tests are still necessary to ensure correct feature implementation.

b) No compiler validation: A fully verified toolchain requires certification that the compiled assembly code correctly implements the uncompiled source. Functional testing informally tests this; static analysis cannot.

c) No higher order logic validation: Many security vulnerabilities are the result of insecure designs, not insecure implementations. Algorithm, protocol, and architecture analysis are outside the scope of static analysis tools.

C. Secure Software Development Process

A complete secure software development process includes many steps in addition to static analysis. Microsoft has developed a detailed sequence of steps called the “Microsoft Software Development Lifecycle” (or SDL). The complete lifecycle is shown in figure 1.

The SDL contains five development stages: requirements, design, implementation, verification, and release. The security focus of each stage is shown in table I.

Stage	**Security Aspects**
Requirements	Security requirements; define bug tracking process
Design	Consult advisors on security; minimize attack surface; perform threat modeling
Implementation	Specify toolchain and flags; run static analysis; minimize bad APIs
Verification	Dynamic analysis; fuzz testing; penetration testing
Release	Document incident response procedure; final security review

TABLE I: Security Aspects of SDL Stages

These stages demonstrate that static analysis is only one tool among many. As Flawfinder’s homepage says, “No tool can substitute for human thought! ... ‘a fool with a tool is still a fool’. It’s a mistake to think that analysis tools (like flawfinder) are a substitute for security training and knowledge” [1]. Static analysis tools supplement training, disciplined bug tracking and incident response, threat modeling, dynamic testing, fuzz testing, penetration testing, and holistic reviews.

The SDL stages are based on the waterfall approach to software development. However, the concepts can be applied to agile if the cycle times are compressed from project-scale to sprint-scale.

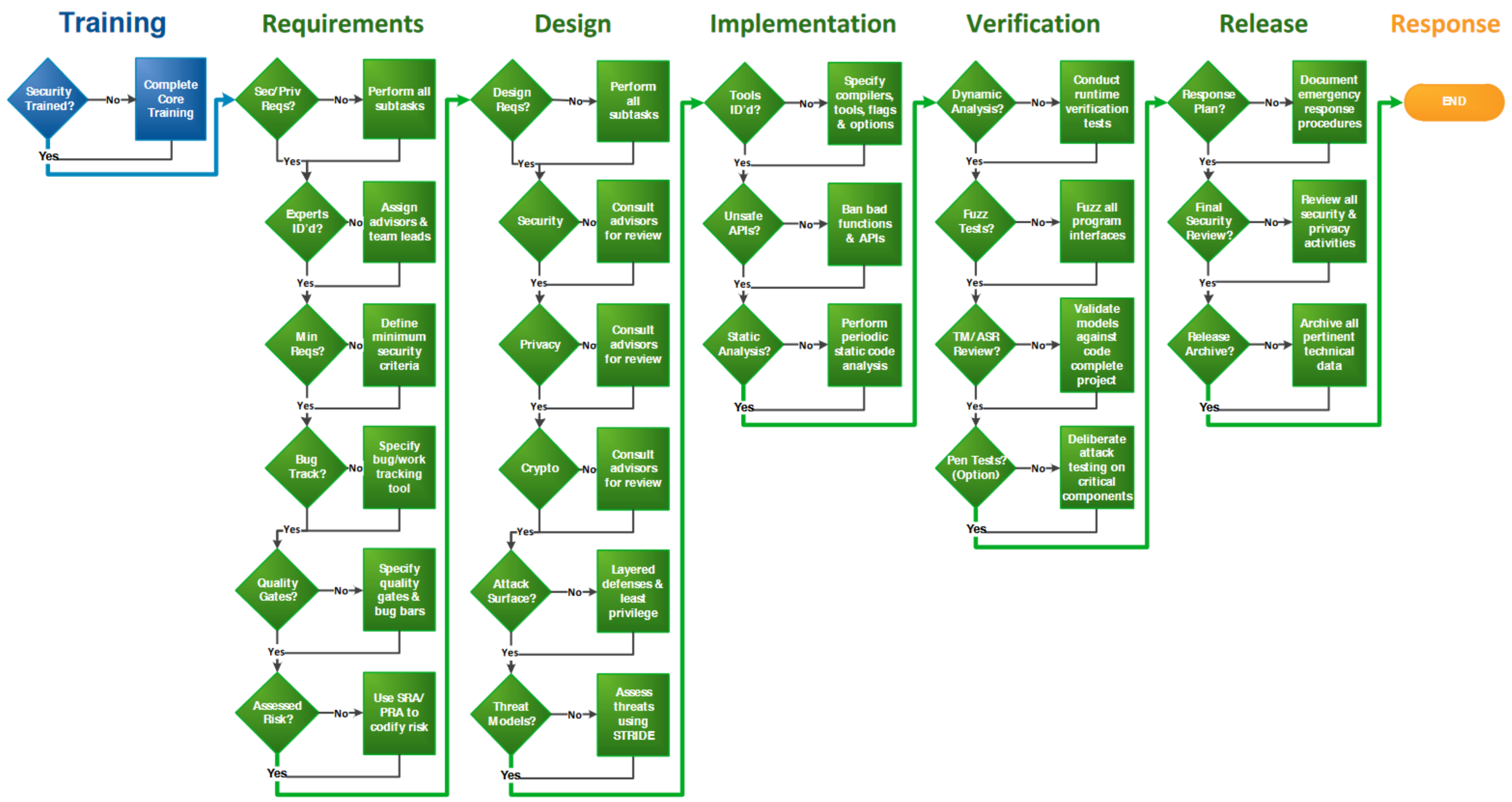
Training
Requirements
Design
Implementation
Verification
Release
Response
Security Trained?
Complete Core Training
Sec/Priv Reqs?
Perform all subtasks
Experts ID'd?
Assign advisors & team leads
Min Reqs?
Define minimum security criteria
Bug Track?
Specify bug/work tracking tool
Quality Gates?
Specify quality gates & bug bars
Assessed Risk?
Use SRA/ PRA to codify risk
Design Reqs?
Perform all subtasks
Security
Consult advisors for review
Privacy
Consult advisors for review
Crypto
Consult advisors for review
Attack Surface?
Layered defenses & least privilege
Threat Models?
Assess threats using STRIDE
Tools ID'd?
Specify compilers, tools, flags & options
Unsafe APIs?
Ban bad functions & APIs
Static Analysis?
Perform periodic static code analysis
Dynamic Analysis?
Conduct runtime verification tests
Fuzz Tests?
Fuzz all program interfaces
TM/ASR Review?
Validate models against code complete project
Pen Tests? (Option)
Deliberate attack testing on critical components
Response Plan?
Document emergency response procedures
Final Security Review?
Review all security & privacy activities
Release Archive?
Archive all pertinent technical data
END
Yes
No

Fig. 1: The Microsoft Software Development Lifecycle Process [2]

III. Flawfinder

The above section discusses static analysis tools in general. This section analyzes one such tool, Flawfinder.

A. *Purpose*

Flawfinder is a free and open source C and C++ static analysis tool licensed under the GNU Public License version 2. The project is defined by simplicity. It is simple in concept, simple in implementation, and simple to use.

Flawfinder focuses on just one of static analysis tools' capabilities listed above–to identify dangerous method usage [1]. C and C++ standard libraries contain many functions with potential security vulnerabilities. The languages are memory unsafe and do not enforce robust error handling. Further, they are used in many security-critical operating system settings. Flawfinder's simple, core purpose is to identify usage of these dangerous methods, rank each instance's risk, and present an ordered list of potential vulnerabilities to the developer. This is its entire function.

Flawfinder is also simple in implementation. Though it analyzes C and C++, it is written in Python. Python is a more expressive language with more powerful string parsing capabilities than C or C++. Python is not as performant as C or C++, but at 45,000 lines/second on 2014 hardware, it is as fast as it needs to be. Lastly, Python has a mature environment of libraries and developers, which means that the runtime is present by default on most machines and future generations can take over development when David Wheeler, the project's main author, no longer personally maintains the project.

Finally, Flawfinder is designed to be simple to use. It can be installed and ran using the commands in listing 1.

```
1 pip install flawfinder
2 flawfinder sourceFile.c
3 flawfinder sourceDir/
```

Listing 1: Flawfinder Basic Usage

If Python is already installed, installing Flawfinder requires a single command, shown in line 1. Flawfinder can be run on a single file as shown in line 2, or recursively on a directory as shown in line 3.

B. *How It Works*

Flawfinder analysis runs in a basic sequence outlined in the following sections.

1) Load a Database of Functions: First, Flawfinder loads a database of error-prone or insecure C and C++ standard library functions. Functions may end up in this list because they are prone to "buffer overflow risks, format string problems, race conditions, potential shell metacharacter dangers, and poor random number acquisition" [1]. This database contains information on the function name, the type of risk it poses, and the magnitude of that risk. The database itself is valuable tool in its own right.

2) Find Function Hits: Second, Flawfinder searches the source code for instances of the functions in the database. This search is context aware–it doesn't do full language parsing, but it does differentiate between code, comments and strings. This makes it smarter than a simple search tool, like `grep`. Flawfinder will also ignore hits in lines that include the comment `/* Flawfinder: ignore */`. This is so that developers can silence warnings for cases that they have analyzed and confirmed to not be real issues.

3) Produce a Report of Hits Sorted by Risk: Third, Flawfinder sorts the hits according to risk and prints a report for the user. The risk and description of the possible vulnerability are other fields in the database. The format of the report can be set using command line arguments. The user can tune the sensitivity to false positives, set a severity reporting threshold, and specify text formatting options.

C. *Example*

An example will demonstrate Flawfinder in action. Listing 2 shows a small C program.

```c
#include <stdio.h>
#include <stdlib.h>

int getNumber() {
    char numberBuffer[10];
    gets(numberBuffer);
    return atoi(numberBuffer);
}

int main() {
    printf("This program multiplies two "
           "numbers.\n");

    printf("Enter number one: ");
    int m1 = getNumber();
    printf("Enter number two: ");
    int m2 = getNumber();

    printf("\nThe product of %d and %d is %d\n",
           m1, m2, m1 * m2);

    return 0;
}
```

Listing 2: C Example Program

This program has a few obvious problems in the `getNumber` function. First, it uses a fixed-size buffer for user input. This might not matter if the input is handled correctly, but second, it uses `gets` to read from `stdin`. `gets` does not check for buffer overflows. Third, it uses `atoi` without checking for error conditions. Aside from security implications, this program may simply crash or produce illogical results without doing more to validate the input.

Running Flawfinder on this file produces the results in listing 3.

```
./example.c:6:  [5] (buffer) gets:
 Does not check for buffer overflows (CWE-120,
   CWE-20). Use fgets() instead.
./example.c:5:  [2] (buffer) char:
 Statically-sized arrays can be improperly
   restricted, leading to potential
 overflows or other issues (CWE-119:CWE-120).
   Perform bounds checking, use
 functions that limit length, or ensure that the
   size is larger than the
 maximum possible length.
./example.c:7:  [2] (integer) atoi:
 Unless checked, the resulting number can exceed
   the expected range
 (CWE-190). If source untrusted, check both
   minimum and maximum, even if the
 input had no minus sign (large numbers can roll
   over into negative number;
 consider saving to an unsigned value if that is
   intended).
```

Listing 3: Example Flawfinder Results

Flawfinder identified the same three problems. Instead of reporting them in order of occurrence, it reported them in order of severity–`gets` is a particularly dangerous method, so it gets a 5. (The severity scale goes from 1-5.) Using a statically sized buffer can be dangerous, and indeed is in this program. However, Flawfinder is does not do thorough code analysis to tell whether the buffer is used safely or not. Because statically allocated buffers are commonly needed and used correctly, it only gets a 2. Finally, it points out that `atoi` is easy to use incorrectly. This also receives a 2.

It is worth noting two points here. First, not all hits are vulnerabilities. Flawfinder will report all instances of static buffers and `atoi` even when they are used properly. In these situations, the `/* Flawfinder ignore */` directive is the developer's friend. Second, the report notes can be quite helpful. The comments are concise, describe the issues, and in some cases direct the developer to safer alternatives, such as `fgets`. This is a strength of the tool.

Full source for the example file and Flawfinder output is available online [3].

IV. Easy Static Analysis

Static analysis tools offer measurable value to a project, but learning each tool and managing install dependencies can take a lot of work. Frameworks and containers offer solutions that can lower the barrier to entry.

A. Lower the Barrier to Entry

Flawfinder is a useful tool, but only exercises a subset of static analysis tools' potential capabilities, and only for C and C++. In order to have a well-rounded static analysis approach, more is needed. More tools means more maintenance and configuration management. Static analysis frameworks address this problem.

Statick [4] is one such framework. Also written in Python, it analyzes a workspace and applies multiple tools according to a detailed configuration or reasonable defaults. Statick's default setting auto-detects files and each relevant tool to it, depending on language. One of the tools it applies is Flawfinder. The same file that was analyzed directly with Flawfinder was also analyzed with Statick using the commands in listing 4. The results are available on GitHub [3].

```
$ mkdir -p statick_output
$ statick . statick_output
```

Listing 4: Example Statick Commands

Statick is not the focus of this paper, so its implementation is not studied in greater depth.

B. Docker Container

Statick solves the problem of static analysis tool configuration, but not environment and dependency management. Docker is a container technology that isolates runtime environments for consistency and portability. It is a perfect tool to use to solve the environment management problem for Statick.

The tools to build a Statick Docker container are available on GitHub [5]. After the user has installed Docker, listing 5 shows all the steps required to build and run the container.

```
$ ./buildImage.sh
$ ./runContainer.sh
Welcome to the Statick Docker environment
$ # in the container; the current directory is
    mounted as /host
$ exit
$ # out of the container
$ ./runContainer.sh mkdir statick_output &&
    statick . statick_output
# statick tool executed in container
$ # out of the container
```

Listing 5: Using the Statick Docker Container

V. Conclusion

Static analysis tools provide useful feedback as part of a complete secure software development workflow. They can be utilized early, catch common errors, and have the potential for powerful, time saving analysis.

Flawfinder is one static analysis tool that spots dangerous function usage in C and C++ code. Its simplicity in concept and implementation is a strength that make it simple to use immediately.

Because static analysis tools specialize in scope and language, a well-rounded static analysis suite requires more than one tool. Frameworks such as Statick manage multi-tool configurations, and running Statick from a Docker container manages environment setup and maintenance.

Taken together, these tools can clearly and quickly identify and reduce software vulnerabilities.

References

[1] D. Wheeler, ”Flawfinder Home Page”, *Dwheeler.com*, 2019. [Online]. Available: https://dwheeler.com/Flawfinder/. [Accessed: 02- Nov- 2019].

[2] ”Simplified Implementation of the Microsoft SDL”, *Microsoft.com*, 2010. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=12379. [Accessed: 13- Nov- 2019].

[3] ”kkredit/static-analysis-research-paper”, *GitHub*, 2019. [Online]. Available: https://github.com/kkredit/static-analysis-research-paper. [Accessed: 18- Nov- 2019].

[4] ”sscpac/statick”, *GitHub*, 2019. [Online]. Available: https://github.com/sscpac/statick. [Accessed: 02- Nov- 2019].

[5] ”kkredit/statick-docker”, *GitHub*, 2019. [Online]. Available: https://github.com/kkredit/statick-docker. [Accessed: 24- Nov- 2019].