Manual vs. Automated Vulnerability Assessment: A Case Study

James A. Kupsch and Barton P. Miller
Computer Sciences Department, University of Wisconsin, Madison, WI, USA
{kupsch,bart}@cs.wisc.edu

Abstract

The dream of every software development team is to assess the security of their software using only a tool. In this paper, we attempt to evaluate and quantify the effectiveness of automated source code analysis tools by comparing such tools to the results of an in-depth manual evaluation of the same system. We present our manual vulnerability assessment methodology, and the results of applying this to a major piece of software. We then analyze the same software using two commercial products, Coverity Prevent and Fortify SCA, that perform static source code analysis. These tools found only a few of the fifteen serious vulnerabilities discovered in the manual assessment, with none of the problems found by these tools requiring a deep understanding of the code. Each tool reported thousands of defects that required human inspection, with only a small number being security related. And, of this small number of security-related defects, there did not appear to be any that indicated significant vulnerabilities beyond those found by the manual assessment.

1 Introduction

While careful design practices are necessary to the construction of secure systems, they are only part of the process of designing, building, and deploying such a system. To have high confidence in a system's security, a systematic assessment of its security is needed before deploying it. Such an assessment, performed by an entity independent of the development team, is a crucial part of development of any secure system. Just as no serious software project would consider skipping the step of having their software evaluated for correctness by an independent testing group, a serious approach to security requires independent assessment for vulnerabilities. At the present time, such an assessment is necessarily an expensive task as it involves a significant commitment of time from a security analyst. While using automated tools is an attractive approach to making this task less labor intensive, even the best of these tools appear limited in the kinds of vulnerabilities that they can identify.

In this paper, we attempt to evaluate and quantify the effectiveness of automated source code vulnerability assessment tools [8] by comparing such tools to the results of an in-depth manual evaluation of the same system.

We started with a detailed vulnerability assessment of a large, complex, and widely deployed distributed system called Condor [12, 15, 2]. Condor is a system that allows the scheduling of complex tasks over local and widely distributed networks of computers that span multiple organizations. It handles scheduling, authentication, data staging, failure detection and recovery, and performance monitoring. The assessment methodology that we developed, called *First Principles Vulnerability Assessment* (FPVA), uses a top-down resource centric approach to assessment that attempts to identify the components of a systems that are most at risk, and then identifying vulnerabilities that might be associated with them. The result of such an approach is to focus on the places in the code where high value assets might be attacked (such as critical configuration files, parts of the code that run at high privilege, or security resources such as digital certificates). This approach shares many characteristics with techniques such as Microsoft's threat modeling [14] but with a key difference: we start from high valued assets and work outward to derive vulnerabilities rather than start with vulnerabilities and then see if they lead to a serious exploit.

In 2005 and 2006, we performed an analysis on Condor using FPVA, resulting in the discovery of fifteen major vulnerabilities. These vulnerabilities were all confirmed by developing sample exploit code that could trigger each one.

More recently, we made an informal survey of security practitioners in industry, government, and academia to identify what were the best automated tools for vulnerability assessment. Uniformly, the respondents identified two highly-regarded commercial tools: Coverity Prevent [4] and Fortify Source Code Analyzer (SCA) [5] (while these companies have multiple products, in the remainder of this paper we will refer to Coverity Prevent and Fortify Source Code Analyzer as "Coverity" and "Fortify" respectively). We applied these tools to the same version of Condor as was used in the FPVA study to compare the ability of these tools to find serious vulnerabilities (having a low false negative rate), while not reporting a significant number of false vulnerabilities or vulnerabilities with limited exploit value (having a low false positive rate).

The most significant findings from our comparative study were:

1. Of the 15 serious vulnerabilities found in our FPVA study of Condor, Fortify found six and Coverity only one.
2. Both Fortify and Coverity had significant false positive rates with Coverity having a lower false positive rate. The volume of these false positives were significant enough to have a serious impact on the effectiveness of the analyst.
3. In the Fortify and Coverity results, we found no significant vulnerabilities beyond those identified by our FPVA study. (This was not an exhaustive study, but did thoroughly cover the problems that the tools identified as most serious.)

To be fair, we did not expect the automated tools to find all the problems that could be found by an experienced analyst using a systematic methodology. The goals of this study were (1) to try to identify the places where an automated analysis can simplify the assessment task, and (2) start to characterize the kind of problems *not* found by these tools so that we can develop more effective automated analysis techniques.

One could claim that the results of this study are not surprising, but there are no studies to provide strong evidence of the strengths and weaknesses of software assessment tools. The contributions of this paper include:

1. showing clearly the limitations of current tools,
2. presenting manual vulnerability assessment as a required part of a comprehensive security audit, and
3. creating a reference set of vulnerabilities to perform apples-to-apples comparisons.

In the next section, we briefly describe our FPVA manual vulnerability assessment methodology, and then in Section 3, we describe the vulnerabilities that were found when we applied FPVA to the Condor system. Next, in Section 4, we describe the test environment in which the automated tools were run and how we applied Coverity and Fortify to Condor. Section 5 describes the results from this study along with a comparison of these results to our FPVA analysis. The paper concludes with comments on how the tools performed in this analysis.

2 First Principles Vulnerability Assessment (FPVA)

This section briefly describes the methodology used to find most of the vulnerabilities used in this study. Most of the vulnerabilities in Condor were discovered using a manual vulnerability assessment we developed at the University of Wisconsin called first principles vulnerability assessment (FPVA). The assessment was done independently, but in cooperation with the Condor development team.

FPVA consists of four analyses where each relies upon the prior steps to focus the work in the current step. The first three steps, architectural, resource, and trust and privilege analyses are designed to assist the assessor in understand the operation of the system under study. The final step, the component evaluation, is where the search for vulnerabilities occurs using the prior analyses and code inspection. This search focuses on likely high-value resources and pathways through the system.

The architectural analysis is the first step of the methodology and is used to identify the major structural components of the system, including hosts, processes, external dependencies, threads, and

major subsystems. For each of these components, we then identify their high-level function and the way in which they interact, both with each other and with users. Interactions are particularly important as they can provide a basis to understand the information flow through, and how trust is delegated through the system. The artifact produced at this stage is a document that diagrams the structure of the system and the interactions.

The next step is the resource analysis. This step identifies the key resources accessed by each component, and the operations supported on these resources. Resources include things such as hosts, files, databases, logs, CPU cycles, storage, and devices. Resources are the targets of exploits. For each resource, we describe its value as an end target (such as a database with personnel or proprietary information) or as an intermediate target (such as a file that stores access-permissions). The artifact produced at this stage is an annotation of the architectural diagrams with resource descriptions.

The third step is the trust and privilege analysis. This step identifies the trust assumptions about each component, answering such questions as how are they protected and who can access them? For example, a code component running on a client's computer is completely open to modification, while a component running in a locked computer room has a higher degree of trust. Trust evaluation is also based on the hardware and software security surrounding the component. Associated with trust is describing the privilege level at which each executable component runs. The privilege levels control the extent of access for each component and, in the case of exploitation, the extent of damage that it can directly accomplish. A complex but crucial part of trust and privilege analysis is evaluating trust delegation. By combining the information from steps 1 and 2, we determine what operations a component will execute on behalf of another component. The artifact produced at this stage is a further labeling of the basic diagrams with trust levels and labeling of interactions with delegation information.

The fourth step is component evaluations, where components are examined in depth. For large systems, a line-by-line manual examination of the code is infeasible, even for a well-funded effort. The step is guided by information obtained in steps 1–3, helping to prioritize the work so that high-value targets are evaluated first. Those components that are part of the communication chain from where user input enters the system to the components that can directly control a strategic resource are the components that are prioritized for assessment. There are two main classifications of vulnerabilities: design (or architectural) flaws, and implementation bugs [13]. Design flaws are problems with the architecture of the system and often involve issues of trust, privilege, and data validation. The artifacts from steps 1–3 can reveal these types of problems or greatly narrow the search. Implementation bugs are localized coding errors that can be exploitable. Searching the critical components for these types of errors results in bugs that have a higher probability of exploit as they are more likely to be in the chain of processing from users input to critical resource. Also the artifacts aid in determining if user input can flow through the implementation bug to a critical resource and allow the resource to be exploited.

3 Results of the Manual Assessment

Fifteen vulnerabilities in the Condor project had been discovered and documented in 2005 and 2006. Most of these were discovered through a systematic, manual vulnerability assessment using the FPVA methodology, with a couple of these vulnerabilities being reported by third parties. Table 1 lists each vulnerability along with a brief description. A complete vulnerability report that includes full details of each vulnerability is available from the Condor project [3] for most of the vulnerabilities.

The types of problems discovered included a mix of implementation bugs and design flaws. The following vulnerabilities are caused by implementation bugs: CONDOR-2005-0003 and CONDOR-2006-000{1,2,3,4,8,9}. The remaining vulnerabilities are caused by design flaws. The vulnerability CONDOR-2006-0008 is unusual in that it only exists on certain older platforms that only provide an unsafe API to create a temporary file.

Table 1: Summary of Condor vulnerabilities discovered in 2005 and 2006 and whether Fortify or Coverity discovered the vulnerability.

Vuln. Id	Fortify	Coverity	Vulnerability Description	Tool Discoverable?
CONDOR-2005-0001	no	no	A path is formed by concatenating three pieces of user supplied data with a base directory path to form a path to to create, retrieve or remove a file. This data is used as is from the client which allows a directory traversal [10] to manipulate arbitrary file locations.	Difficult. Would have to know path was formed from untrusted data, not validated properly, and that a directory traversal could occur. Could warn about untrusted data used in a path.
CONDOR-2005-0002	no	no	This vulnerability is a lack of authentication and authorization. This allows impersonators to manipulate checkpoint files owned by others.	Difficult. Would have to know that there should be an authentication and authorization mechanism, which is missing.
CONDOR-2005-0003	yes	no	This vulnerability is a command injection [10] resulting from user supplied data used to form a string. This string is then interpreted by `/bin/sh` using a `fork` and `execl("/bin/sh", "-c", command)`.	Easy. Should consider network and file data as tainted and all the parameters to execl as sensitive.
CONDOR-2005-0004	no	no	This vulnerability is caused by the insecure owner of a file used to store persistent overridden configuration entries. These configuration entries can cause arbitrary executable files to be started as root.	Difficult. Would have to track how these configuration setting flow into complex data structure before use, both from files that have the correct ownership and permissions and potentially from some that do not.
CONDOR-2005-0005	no	no	This vulnerability is caused by the lack of an integrity [10] check on checkpoints (a representation of a running process that can be restarted) that are stored on a checkpoint server. Without a way of ensuring the integrity of the checkpoint, the checkpoint file could be tampered with to run malicious code.	Difficult. This is a high level design flaw that a particular server should not be trusted.
CONDOR-2005-0006	no	no	Internally the Condor system will not run user's jobs with the user id of the root account. There are other accounts on machines which should also be restricted, but there are no mechanisms to support this.	Difficult. Tool would have to know which accounts should be allowed to be used for what purposes.
CONDOR-2006-0001	yes	no	The stork subcomponent of Condor, takes a URI for a source and destination to move a file. If the destination file is local and the directory does not exist the code uses the `system` function to create it without properly quoting the path. This allows a command injection to execute arbitrary commands. There are 3 instances of this vulnerability.	Easy. The string used as the parameter to `system` comes fairly directly from an untrusted argv value.
CONDOR-2006-0002	yes	no	The stork subcomponent of Condor, takes a URI for a source and destination to move a file. Certain combinations of schemes of the source and destination URIs cause stork to call helper applications using a string created with the URIs, and without properly quoting them. This string is then passed to `popen`, which allows a command injection to execute arbitrary commands. There are 6 instances of this vulnerability.	Easy. The string used as the parameter to `popen` comes from a substring of an untrusted `argv` value.

Table 1 – Continued.

Vuln. Id	Fortify	Coverity	Vulnerability Description	Tool Discoverable?
CONDOR-2006-0003	yes	no	Condor class ads allow functions. A function that can be enabled, executes an external program whose name and arguments are specified by the user. The output of the program becomes the result of the function. The implementation of the function uses `popen` without properly quoting the user supplied data.	Easy. A call to `popen` uses data from an untrusted source such as the network or a file.
CONDOR-2006-0004	yes	no	Condor class ads allow functions. A function that can be enabled, executes an external program whose name and arguments are specified by the user. The path of the program to run is created by concatenating the script directory path with the name of the script. Nothing in the code checks that the script name cannot contain characters that allows for a directory traversal.	Easy. A call to `popen` uses data from an untrusted source such as the network or a file. It would be difficult for a tool to determine if an actual path traversal is possible.
CONDOR-2006-0005	no	no	This vulnerability involves user supplied data being written as records to a file with the file later reread and parsed into records. Records are delimited by a new line, but the code does not escape new lines or prevent them in the user supplied data. This allows additional records to be injected into the file.	Difficult. Would have to deduce the format of the file and that the injection was not prevented.
CONDOR-2006-0006	no	no	This vulnerability involves an authentication mechanism that assumes a file with a particular name and owner can be created only by the owner or the root user. This is not true as any user can create a hard link, in a directory they write, to any file and the file will have the permissions and owner of the linked file, invalidating this assumption.[11]	Difficult. Would require the tool to understand why the existence and properties are being checked and that they can be attacked in certain circumstances.
CONDOR-2006-0007	no	no	This vulnerability is due to a vulnerability in OpenSSL [9] and requires a newer version of the library to mitigate.	Difficult. The tool would have to have a list of vulnerable library versions. It would also be difficult to discover if the tool were run on the library code as the defect is algorithmic.
CONDOR-2006-0008	no	no	This vulnerability is caused by using a combination of the functions `tmpnam` and `open` to try and create a new file. This allows an attacker to use a classic time of check, time of use (TOCTOU) [10] attack against the program to trick the program into opening an existing file. On platforms that have the function `mkstemp`, it is safely used instead.	Hard. The unsafe function is only used (compiled) on a small number of platforms. This would be easy for a tool to detect if the unsafe version is compiled. Since the safe function `mkstemp` existed on the system, the unsafe version was not seen by the tools.
CONDOR-2006-0009	yes	yes	This vulnerability is caused by user supplied values being placed in a fixed sized buffer that lack bounds checks. The user can then cause a buffer overflow [16] that can result in a crash or stack smashing attack.	Easy. No bounds check is performed when writing to a fixed sized buffer (using the dangerous function `strcpy`) and the data comes from an untrusted source.
Total	6	1	out of 15 total vulnerabilities	

4 Setup and Running of Study

4.1 Experiment Setup

To perform the evaluation of the Fortify and Coverity tools, we used the same version of Condor, run in the same environment, as was used in our FPVA analysis. The version of the source code, platform

and tools used in this test were as follows:

1. Condor 6.7.12
   (a) with 13 small patches to allow compilation with newer GNU compiler collection (gcc) [6];
   (b) built as a clipped [1] version, i.e., no standard universe, Kerberos, or Quill as these would not build without extensive work on the new platform and tool chain.
2. gcc (GCC) 3.4.6 20060404 (Red Hat 3.4.6-10)
3. Scientific Linux SL release 4.7 (Beryllium) [7]
4. Fortify SCA 5.1.0016 rule pack 2008.3.0.0007
5. Coverity Prevent 4.1.0

To get both tools to work required using a version of gcc that was newer than had been tested with Condor 6.7.12. This necessitated 13 minor patches to prevent gcc from stopping with an error. Also this new environment prevented building Condor with standard universe support, Kerberos, and Quill. None of these changes affected the presence of the discovered vulnerabilities.

The tools were run using their default settings except Coverity was passed the flag `--all` to enable all the analysis checkers (Fortify enables all by default).

4.2 Tool Operation

Both tools operate in a similar three step fashion: gather build information, analyze, and present results. The build information consists of the files to compile, and how they are used to create libraries and executable files. Both tools make this easy to perform by providing a program that takes as arguments the normal command used to build the project. The information gathering tool monitors the build's commands to create object files, libraries and executables.

The second step performs the analysis. This step is also easily completed by running a program that takes the result of the prior step as an input. The types of checkers to run can also be specified. The general term *defect* will be used to describe the types of problems found by the tools as not all problems result in a vulnerability.

Finally, each tool provides a way to view the results. Coverity provides a web interface, while Fortify provides a stand-alone application. Both viewers allow the triage and management of the discovered defects. The user can change attributes of the defect (status, severity, assigned developer, etc.) and attach additional information. The status of previously discovered defects in earlier analysis runs is remembered, so the information does not need to be repeatedly entered.

Each tool has a collection of checkers that categorize the type of defects. The collection of checkers depends on the source language and the options used during the analysis run. Fortify additionally assigns each defect a severity level of Critical, Hot, Warning and Info. Coverity does not assign a severity, but allows one to be assigned by hand.

4.3 Tool Output Analysis

After both tools were run on the Condor source, the results from each tool were reviewed against the known vulnerabilities and were also sampled to look for vulnerabilities that were not found using the FPVA methodology.

The discovered vulnerabilities were all caused by code at one or at most a couple of a lines or functions. Both tools provided interfaces that allowed browsing the found defects by file and line. If the tool reported a defect at the same location in the code and of the correct type the tool was determined to have found the vulnerability.

The defects discovered by the tools were also sampled to determine if the tools discovered other vulnerabilities and to understand the qualities of the defects. The sampling was weighted to look more at defects found in higher impact locations in the code and in the categories of defects that are more likely to impact security. We were unable to conduct an exhaustive review the results due to time constraints and the large number of defects presented by the tools.

5 Results of the Automated Assessment

This section describes the analysis of the defects found by Coverity and Fortify. We first compare the results of the tools to the vulnerabilities found by FPVA. Next we empirically look at the false positive and false negative rates of the tools and the reasons behind these. Finally we offer some commentary on how the tools could be improved.

Fortify discovered all the vulnerabilities we expected it to find, those caused by implementation bugs, while Coverity only found a small subset. Each tool reported a large number of defects. Many of these are indications of potential correctness problems, but out of those inspected none appeared to be a significant vulnerability.

5.1 Tools Compared to FPVA Results

Table 1 presents each vulnerability along with an indication if Coverity or Fortify also discovered the vulnerability.

Out of the fifteen known vulnerabilities in the code, Fortify found six of them, while Coverity only discovered one of them. Vulnerability CONDOR-2006-0001 results from three nearly identical vulnerability instances in the code, and vulnerability CONDOR-2006-0002 results from six nearly identical instances. Fortify discovered all instances of these two vulnerabilities, while Coverity found none of them.

All the vulnerabilities discovered by both tools were due to Condor's use of functions that commonly result in security problems such as `execl`, `popen`, `system` and `strcpy`. Some of the defects were traced to untrusted inputs being used in these functions. The others were flagged solely due to the dangerous nature of these functions. These vulnerabilities were simple implementation bugs that could have been found by using simple scripts based on tools such as `grep` to search for the use of these functions.

5.2 Tool Discovered Defects

Table 2 reports the defects that we found when using Fortify, dividing the defects into categories with a count of how often each defect category occurred. Table 3 reports the defects found when using Coverity. The types of checkers that each tool reports are not directly comparable, so no effort was done to do so. Fortify found a total of 15,466 defects while Coverity found a total of 2,686. The difference in these numbers can be attributed to several reasons:

1. differences in the analysis engine in each product;
2. Coverity creates one defect for each sink (place in the code where bad data is used in a way to cause the defect, and displays one example source to sink path), while Fortify has one defect for each source/sink pair; and
3. Coverity seems to focus on reducing the number of false positives at the risk of missing true positives, while Fortify is more aggressive in reporting potential problems resulting in more false positives.

From a security point of view, the sampled defects can be categorized in order of decreasing importance as follows:

1. *Security Issues.* These problems are exploitable. Other than the vulnerabilities also discovered in the FPVA (using tainted data in risk functions), the only security problems discovered were of a less severe nature. They included denial of service issues due to the dereference of null pointers, and resource leaks.
2. *Correctness Issues.* These defects are those where the code will malfunction, but the security of the application is not affected. These are caused by problems such as (1) a buffer overflow of a small number of bytes that may cause incorrect behavior, but do not allow execution of arbitrary code or other security problems, (2) the use of uninitialized variables, or (3) the failure to check the status of certain functions.

3. *Code Quality Issues.* Not all the defects found are directly security related, such as Coverity's parse warnings (those starting with PW), dead code and unused variables, but they are a sign of code quality and can result in security problem in the right circumstances.

Due to the general fragility of code, small changes in code can easily move a defect from one category to another, so correcting the non-security defects could prevent future vulnerabilities.

5.3 False Positives

False positives are the defects that the tool reports, but are not actually defects. Many of these reported defects are items that should be repaired as they often are caused by poor programming practices that can easily develop into a true defect during modifications to the code. Given the finite resources in any assessment activity, these types of defects are rarely fixed. Ideally, a tool such as Fortify or Coverity is run regularly during the development cycle, allowing the programmers to fix such defects as they appear (resulting in a lower false positive rate). In reality, these tools are usually applied late in the lifetime of a software system.

Some of the main causes of false positives found in this study are the following:

1. Non-existent code paths due to functions that never return due to an `exit` or `exec` type function. Once in a certain branch, the program is guaranteed to never execute any more code in the program due to these functions and the way that code is structured, but the tool incorrectly infers that it can continue past this location.
2. Correlated variables, where the value of one variable restricts the set of values the other can take. This occurs when a function returns two values, or two fields of a structure. For instance, a function could return two values, one a pointer and the other a boolean indicating that the pointer is valid; if the boolean is checked before the dereferencing of the pointer, the code is correct, but if the tool does not track the correlation it appears that a null pointer dereference could occur.
3. The value of a variable is restricted to a subset of the possible values, but is not deduced by the tool. For instance, if a function can return only two possible errors, and a switch statement only handles these exact two errors, the code is correct, but a defect is produced due to not all possible errors being handled.
4. Conditions outside of the function prevent a vulnerability. This is caused when the tool does not deduce that:
   (a) Data read from certain files or network connections should be trusted due to file permissions or prior authentication.
   (b) The environment is secure due to a trusted parent process securely setting the environment.
   (c) A variable is constrained to safe values, but it is hard to deduce.

The false positives tend to cluster in certain checkers (and severity levels in Fortify). Some checkers will naturally have less reliability than others. The other cause of the cluster is due to developers repeating the same idiom throughout the code. For instance, almost all of the 330 UNINIT defects that Coverity reports are false positives due to a recurring idiom.

Many of these false positive defects are time bombs waiting for a future developer to unwittingly make a change somewhere in the code that affects the code base to now allow the defect to be true. A common example of this is a string buffer overflow, where the values placed in the buffer are currently too small in aggregate to overflow the buffer, but if one of these values is made bigger or unlimited in the future, the program now has a real defect.

Many of the false positives can be prevented by switching to a safer programming idiom, where it should take less time to make this change than for a developer to determine if the defect is actually true or false. The uses of `sprintf`, `strcat` and `strcpy` are prime examples of this.

Table 2: Defect counts reported by Fortify by type and severity level.

Vuln Type	Total	Critical	Hot	Warning	Info
Buffer Overflow	2903	0	1151	391	1361
Buffer Overflow: Format String	1460	0	995	465	0
Buffer Overflow: Format String (%f/%F)	75	0	42	33	0
Buffer Overflow: Off-by-One	4	0	4	0	0
Command Injection	108	0	81	15	12
Dangerous Function	3	3	0	0	0
Dead Code	589	0	0	0	589
Denial of Service	2	0	0	2	0
Double Free	33	0	0	33	0
Format String	105	0	27	24	54
Format String: Argument Type Mismatch	3	0	0	3	0
Heap Inspection	16	0	0	0	16
Illegal Pointer Value	1	0	0	0	1
Insecure Randomness	5	0	0	0	5
Insecure Temporary File	6	0	0	1	5
Integer Overflow	1168	0	0	274	894
Memory Leak	906	0	0	906	0
Memory Leak: Reallocation	6	0	0	6	0
Missing Check against Null	670	0	0	670	0
Null Dereference	263	0	0	263	0
Obsolete	78	0	0	0	78
Often Misused: Authentication	24	0	0	0	24
Often Misused: File System	5	0	0	0	5
Often Misused: Privilege Management	15	0	0	0	15
Out-of-Bounds Read	2	0	0	2	0
Out-of-Bounds Read: Off-by-One	3	0	0	3	0
Path Manipulation	463	0	0	444	19
Poor Style: Redundant Initialization	14	0	0	0	14
Poor Style: Value Never Read	120	0	0	0	120
Poor Style: Variable Never Used	277	0	0	0	277
Process Control	1	0	1	0	0
Race Condition: File System Access	92	0	0	92	0
Race Condition: Signal Handling	15	0	0	15	0
Redundant Null Check	108	0	0	108	0
Resource Injection	58	0	0	58	0
Setting Manipulation	28	0	0	28	0
String Termination Error	4708	0	0	3702	1006
System Information Leak	760	0	0	458	302
Type Mismatch: Signed to Unsigned	2	0	0	0	2
Unchecked Return Value	137	0	0	0	137
Uninitialized Variable	125	0	0	0	125
Unreleased Resource	82	0	0	82	0
Unreleased Resource: Synchronization	2	0	0	2	0
Use After Free	21	0	0	21	0
Total	15466	3	2301	8101	5061

Table 3: Defect counts reported by Coverity by type.

Total	Vulnerability Type	Total	Vulnerability Type
2	ARRAY_VS_SINGLETON	38	REVERSE_INULL
1	ATOMICITY	0	REVERSE_NEGATIVE
0	BAD_ALLOC_ARITHMETIC	842	SECURE_CODING
0	BAD_ALLOC_STRLEN	4	SECURE_TEMP
0	BAD_COMPARE	2	SIZECHECK
0	BAD_FREE	0	SLEEP
1	BAD_OVERRIDE	378	STACK_USE
1	BUFFER_SIZE	1	STREAM_FORMAT_STATE
32	BUFFER_SIZE_WARNING	2	STRING_NULL
5	CHAR_IO	147	STRING_OVERFLOW
82	CHECKED_RETURN	10	STRING_SIZE
0	CHROOT	6	TAINTED_SCALAR
2	CTOR_DTOR_LEAK	43	TAINTED_STRING
29	DEADCODE	26	TOCTOU
5	DELETE_ARRAY	0	UNCAUGHT_EXCEPT
0	DELETE_VOID	330	UNINIT
0	EVALUATION_ORDER	96	UNINIT_CTOR
40	FORWARD_NULL	9	UNREACHABLE
2	INFINITE_LOOP	31	UNUSED_VALUE
0	INTEGER_OVERFLOW	12	USE_AFTER_FREE
0	INVALIDATE_ITERATOR	5	VARARGS
0	LOCK	0	WRAPPER_ESCAPE
0	LOCK_FINDER	1	PW.BAD_MACRO_REDEF
3	MISSING_LOCK	5	PW.BAD_PRINTF_FORMAT_STRING
17	MISSING_RETURN	56	PW.IMPLICIT_FUNC_DECL
17	NEGATIVE_RETURNS	1	PW.IMPLICIT_INT_ON_MAIN
18	NO_EFFECT	18	PW.INCLUDE_RECURSION
32	NULL_RETURNS	20	PW.MISSING_TYPE_SPECIFIER
4	OPEN_ARGS	46	PW.NON_CONST_PRINTF_FORMAT_STRING
4	ORDER_REVERSAL	2	PW.PARAMETER_HIDDEN
3	OVERRUN_DYNAMIC	20	PW.PRINTF_ARG_MISMATCH
30	OVERRUN_STATIC	10	PW.QUALIFIER_IN_MEMBER_DECLARATION
3	PASS_BY_VALUE	2	PW.TOO_FEW_PRINTF_ARGS
1	READLINK	7	PW.TOO_MANY_PRINTF_ARGS
150	RESOURCE_LEAK	11	PW.UNRECOGNIZED_CHAR_ESCAPE
0	RETURN_LOCAL	21	PW.USELESS_TYPE_QUALIFIER_ON_RETURN-_TYPE
		2686	Total

5.4 False Negatives

False negatives are defects in the code that the tool did not report. These defects include the following:

1. Defects that are high level design flaws. These are the most difficult defects for a tool to detect as the tool would have to understand design requirements not present in the code.
2. The dangerous code is not compiled on this platform. The tools only analyze the source code seen when the build information gathering step is run. The tools ignore files that were not compiled and parts of files that were conditionally excluded. A human inspecting the code can easily spot problems that occur in different build configurations.
3. Tainted data becomes untainted. The five vulnerabilities that Fortify found, but Coverity did not were caused by Coverity only reporting an issue with functions such as `execl`, `popen` and `system` if the data is marked as tainted. The tainted property of strings is only transitive when calling certain functions such as `strcpy` or `strcat`. For instance, if a substring is copied byte by byte, Coverity does not consider the destination string as tainted.
4. Data flows through a pointer to a heap data structure, that the tool cannot track.

Some of these are defects that a tool will never find, while some of these will hopefully be found by tools in the future as the quality of their analysis improves.

5.5 Improving the Tool's Results

Both tools allow the analyst to provide more information to the tool to increase the tools accuracy. This information is described by placing annotations in the source code, or a simple description of the additional properties can be imported into the tools analysis model.

A simple addition could be made to Coverity's model to flag all uses of certain system calls as unsafe. This would report all the discovered vulnerabilities that Fortify found along with all the false positives for these types of defects.

6 Conclusion

This study demonstrates the need for manual vulnerability assessment performed by a skilled human as the tools did not have a deep enough understanding of the system to discover all of the known vulnerabilities.

There were nine vulnerabilities that neither tools discovered. In our analysis of these vulnerabilities, we did not expect a tool to find them due as they are caused by design flaws or were not present in the compiled code.

Out of the remaining six vulnerabilities, Fortify did find them all, and Coverity found a subset and should be able to find the others by adding a small model. We expected a tool and even a simple to tool to be able to discover these vulnerabilities as they were simple implementation bugs.

The tools are not perfect, but they do provide value over a human for certain implementation bugs or defects such as resource leaks. They still require a skilled operator to determine the correctness of the results, how to fix the problem and how to make the tool work better.

7 Acknowledgments

This research funded in part by National Science Foundation grants OCI-0844219, CNS-0627501, and CNS-0716460.

References

[1] *Condor Manual*. Condor Team, University of Wisconsin. `http://www.cs.wisc.edu/condor/manual`.

[2] Condor Project. `http://www.cs.wisc.edu/condor`.

[3] Condor Vulnerability Reports. `http://www.cs.wisc.edu/condor/security/vulnerabilities`.

[4] Coverity Inc., Prevent. `http://www.coverity.com`.

[5] Fortify Software Inc., Source Code Analyzer (SCA). `http://www.fortify.com`.

[6] GNU Compiler Collection (gcc). `http://gcc.gnu.org`.

[7] Scientific Linux, CERN and Fermi National Accelerator Laboratory. `http://www.scientificlinux.org`.

[8] Brian Chess and Jacob West. *Secure Programming with Static Analysis*. Addison-Wesley, 2007.

[9] CVE-2006-4339. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4339`, 2006. OpenSSL vulnerability.

[10] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2007.

[11] James A. Kupsch and Barton P. Miller. How to Open a File and Not Get Hacked. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 1196–1203, 2008.

[12] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A Hunter of Idle Workstations. *Proc. 8th Intl Conf. on Distributed Computing Systems*, pages 104–111, June 1988.

[13] Gary McGraw. *Software Security*. Addison-Wesley, 2006.

[14] Frank Swiderski and Window Snyder. *Threat Modeling*. Microsoft Press, 2004.

[15] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[16] John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.