First Principles Vulnerability Assessment

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

Elisa Heymann Eduardo César
Computer Arch and Operating Systems Dept
Universitat Autònoma de Barcelona
Bellaterra (Barcelona), SPAIN
{elisa.heymann,eduardo.cesar}@uab.es

ABSTRACT

Clouds and Grids offer significant challenges to providing secure infrastructure software. As part of a our effort to secure such middleware, we present First Principles Vulnerability Assessment (FPVA), a new analyst-centric (manual) technique that aims to focus the analyst's attention on the parts of the software system and its resources that are most likely to contain vulnerabilities that would provide access to high-value assets. FPVA finds new threats to a system and is not dependent on a list of known threats.

Manual assessment is labor-intensive, making the use of automated assessment tools quite attractive. We compared the results of FPVA to those of the top commercial tools, providing the first significant evaluation of these tools against a real-world known collection of serious vulnerabilities. While these tools can find common problems in a program's source code, they miss a significant number of serious vulnerabilities found by FPVA. We are now using the results of this comparison study to guide our future research into improving automated software assessment.

Categories and Subject Descriptors

K.6.5 [**Manage of Computing and Information Systems**]: Security and Protection—*invasive software*

General Terms

Security

Keywords

auditing, tiger team

1. INTRODUCTION

Vulnerability assessment of critical software is repeatedly described as one of the gaps in national and international cyber-defense strategies [3]. Cloud and Grid computing, by their distributed nature, dynamic allocation, and multiple organizations, expose new attack surfaces and provide new opportunities for attacks on both the service and client infrastructures. While automated analysis tools play a key role in the assessment of such systems, the current state of the art for such tools limits their effectiveness.

In this paper we discuss our approach to the assessment of such systems. Our approach starts a new methodology for manual (analyst centric) vulnerability assessment, called *First Principles Vulnerability Assessment* (FPVA). FPVA allows us to evaluate the security of a system in depth. While FPVA is certainly a labor intensive approach to vulnerability assessment, we have shown it to be effective in several real systems, finding many serious vulnerabilities. Many of these vulnerabilities reflect common serious mistakes made in distributed services. These mistakes include erroneous or changeable configuration files, injection attacks and race conditions.

While our use of FPVA has uncovered significant vulnerabilities in a wide variety of complex systems (along with suggested remediations for these vulnerabilities), it is only the starting point. The in-depth FPVA analysis of real systems has produced a body of significant vulnerabilities, and this body of vulnerabilities can act as a *reference set* for evaluating automated analysis tools. Our FPVA assessment results act as an approximation of a ground truth, allowing us to compare the results from our FPVA study to those obtained from running widely used (and highly regarded) commercial source code analysis tools.

Our new assessment methodology and its careful comparison to automated tools are guiding us in two new research directions: reducing the cost of performing a manual assessment, and improving the ability of automated tools to find the most serious vulnerabilities.

FPVA assumes access to the source code, documentation, and, when available, the developers. Rather than working from known vulnerabilities, the starting point for FPVA is to identify *high value assets* in a system, i.e., those components (for example, processes or parts of processes that run with high privilege) and resources (for example, configuration files, databases, connections, and devices) whose exploitation offers the greatest potential for damage by an intruder. From these components and resources, we work outward to discover execution paths through the code that might exploit them. This approach has a couple of advantages. First, it allows us to find new vulnerabilities, not just exploits based on those that were previously discovered. Second, when a vulnerability is discovered, it is likely to be a serious one whose remediation is of high priority.

CCSW'10, October 8, 2010, Chicago, Illinois, USA.

FPVA starts with an architectural analysis of the system, identifying the key components in a distributed system, as well as the interactions among the different components and the interfaces available to users for providing input (attack surface). It then goes on to identify the resources associated with each component, the privilege level of each component, the value of each resource, how the components interact, and how trust is delegated. The results of these steps are documented in clear diagrams that provide a roadmap for the last stage of the analysis, the manual code inspection. In addition, *the results of this step can also form the basis for a risk assessment of the system, identifying which parts of the system are most immediately in need of evaluation.*

After these steps, we then use code inspection techniques on the critical parts of the code. Our analysis strategy targets the high value assets in a system and focuses attention on the parts of the system that are vulnerable to not only unauthorized entry, but unauthorized entry that can be exploited. *Note that we consider a vulnerability to be verified only after we produce actual exploit code for it.*

We have applied FPVA to seven major systems, including the Condor high-throughput scheduling system [13, 16]. In these analyses, significant vulnerabilities were found, software development teams were educated to the key issues, and the resulting software was made more resistant to attack.

A major question when evaluating a technique such as FPVA is: why not use an automated code analysis tool? To address this question, we surveyed security practitioners in academia, industry, and government laboratories to identify which tools were consider "best of breed" in automated vulnerability assessment. Repeatedly, two commercials software packages were named, Coverity Prevent [4] and Fortify Source Code Analyzer (SCA) [7]. To evaluate the power of these tools (and several others) and better understand our new FPVA, we compared the results of our largest assessment activity (on Condor) to the results gathered from applying these automated tools. As a result of these studies, we found that the automated tools found few of the vulnerabilities that we had identified in Condor (i.e., had significant false negatives) and identified many problems that were either not exploitable or led to minor vulnerabilities (i.e., had many false positives).

Our current work is just the start of a longer term effort to develop more effective assessment techniques. We are using our experiences with these techniques to help design tools that will simplify the task of manual assessment. In addition, we are working to develop a formal characterization of the vulnerabilities we have found in an attempt to develop improved automated detection algorithms that would include more of these vulnerabilities. We are also using our experience with these techniques to develop new guidelines for programming techniques that lead to more secure code.

2. FIRST PRINCIPLES VULNERABILITY ASSESSMENT METHODOLOGY

A key benefit of our approach to vulnerability assessment is that we do not make a priori assumptions about the nature of threats or the techniques that will be used to exploit a vulnerability. We can characterize the steps of architectural, resource, privilege, and interaction analysis as a narrowing processing that produces a focused code analysis. In addition, these analysis steps can help to identify more complex vulnerabilities that are based on the interaction of multiple system components and are not amenable to local code analysis. For example, we have seen several real vulnerabilities that are caused because a component (process) is allowed to write a file that will be later read by another component. The read or write operation by itself does not appear harmful, but how the data was created or used in another part of the code could allow an exploit to occur. In the Condor system, both configuration and checkpoint files were vulnerable to such attacks. Without a more global view of the analysis, such problems are difficult to find. Another example involves creating an illegal event that is queued and later processed by another component. In Condor, this situation arose from creating a job-submission record with illegal attributes. There was nothing to indicate that this was a problem at the time the record was created, but it became dangerous at the time (and place) it was used.

We rate security vulnerabilities on a four-level scale:

- Level 0: False alarm: The exploit for this vulnerability does not actually allow any unauthorized access.
- Level 1: Zero-value vulnerability: The exploit allows unauthorized access to the system, but no assets of any value can be accessed.
- Level 2: Low-value asset access: The exploit allows unauthorized access, provides access to an asset (or prevents access to an asset), but is considered a lesser threat. An example of such a vulnerability might be allowing read access to a log file.
- Level 3: High value asset access: The exploit allows unauthorized access, provides access to an asset, and the asset is of a critical nature. An example of such a vulnerability might be allowing unauthorized log-in to a server or revealing critical data.

While there can be a subjective nature to the labeling of the value of assets, in operational practice, there is usually little ambiguity. Our goal is to spend the majority of our time finding and correcting Level 3 vulnerabilities, and to some extent, those at Level 2.

First Principles Vulnerability Assessment is composed of 5 main steps:

Architectural analysis:

The first step is to identify the major structural components of the system, including modules, threads, processes, and hosts. For each of these components, we then identify the way in which they interact, both with each other and with users. Interactions are particularly important as they can provide a basis for understanding how trust is delegated through the system. In this step we also identify the *attack surface*, that is the interfaces available to users for providing input to the system. This is a key aspect as all attacks will come from user supplied data. The artifact produced at this stage is a document that diagrams the structure of the system and the interactions amongst the different components, and with the end users. Figure 1 shows an example of such a diagram.

Resource identification:

The second step is to identify the key resources accessed by each component, and the operations supported on those

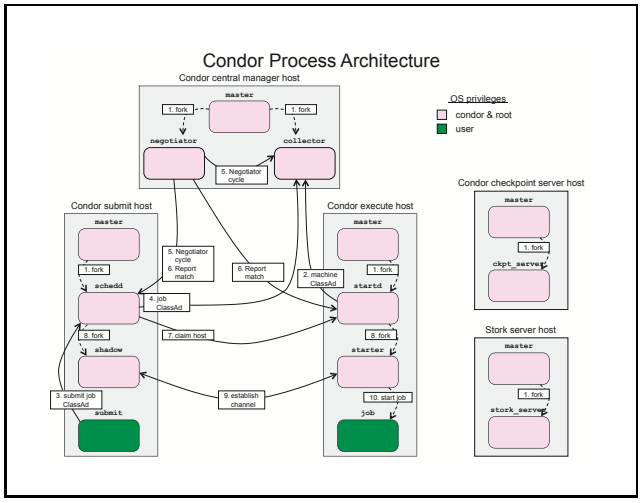

Figure 1: Example Architecture Diagram from Condor. Presents hosts, processes, and the steps required to execute a job when Condor is installed as root.

resources. Resources include data elements such as files, databases, logs, and devices; and physical entities such as CPU cycles and network bandwidth. These resources are often the target of an exploit. 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.

Trust and privilege analysis:

The third 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 accomplish directly. A complex but crucial part of trust and privilege analysis is evaluating trust delegation. By combining the information from the first two steps, 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 coloring of nodes in Figure 1 is an example of the results of this step.

Component evaluation:

The fourth step is to examine each component in depth. For large systems, a line-by-line manual examination of the code is infeasible, even for a well-funded effort. *A key aspect of our technique is that this step is guided by information obtained in the first three steps, helping to prioritize the work so that high-value targets are evaluated first.* The work in this step can be accelerated by automated scanning tools. While these tools can provide valuable information, they are subject to false positives, and even when they indicate real flaws, they often cannot tell whether the flaw is exploitable and, even if it is, whether it will allow serious damage. In addition, these tools typically work most effectively on a local basis, so flaws based on inappropriate trust boundaries or delegation of authority are not likely to be found. Therefore, these tools work best in the context of a vulnerability analysis process. We describe our experience with such tools in more detail in Section 4. The artifacts produced by this step are vulnerability reports, perhaps with suggested fixes, to be provided to the software developers.

A key question in the component analysis is when to stop? In other words, how much analysis is enough? Exhaustive formal analysis of the code is typical not affordable, let alone feasible, so the amount of time spent on an assessment becomes a cost-benefit trade-off. At some point in the analysis, a careful examination will be completed on the code involving the high-value assets that are reachable via the attack surface. When this point is reached, then subsequent vulnerabilities that are found are likely to be of lesser significance.

Dissemination of results:

Once vulnerabilities are reported, the obvious next step is for the developers to fix them. However, once fixed, they are confronted with some questions that can be difficult to answer in a collaborative and often open-source world. Some of these include: How do we integrate the update into our release stream? (Do we put out a special release or part of an upcoming one?) When do we announce the existence of the vulnerability? How much detail do we provide initially? If the project is open source, how do we deal with groups that are slow to update? Should there be some community-wide mechanism to time announcements and releases? Due to a lack of space we do not answer these questions in this paper, but we refer to the FPVA whitepaper [12] where we address the issue of integrating vulnerability assessment into the software development cycle.

3. SUMMARY OF RESULTS

The FPVA methodology was developed in response to a need for an effective in-depth assessment methodology. As we studied our first systems, we codified what worked and continued to refine these techniques as we assessed additional systems. As a result, we believe that we have a reasonably mature approach that has resulted in several benefits. First, as described in this section, we have made concrete contributions to improving the security of several critical middleware systems. Second, as described in Section 4, our results have provided the foundation for a careful study of the effectiveness of automated analysis tools. Third, our techniques and results have laid the foundation for our ongoing research agenda based on developing formal characterizations (and detection algorithms) for the more complex vulnerabilities that we have found and for automating key parts of the FPVA methodology.

One area of our success has been to apply FPVA to a variety of Grid middleware. Cloud computing environments share many of the same features and goals as Grid computing, and we are starting new assessments in this area. Here we report on our application of FPVA to seven different systems written by different authors, and the assessments

Table 1: Worst-case impact of defects discovered by project. Impacts allowed access to accounts on the system or allowed a denial of service (DOS).

Project	Root Acct	Admin Acct	Other Acct	DOS	Total
Condor	1	13	1		15
SRB		5			5
MyProxy		1	1	3	5
gLExec	3	1	1		5
Gratia Condor Probe	2			1	3
Condor Quill				6	6
CrossBroker			2	1	3
Total	6	20	5	11	42

carried out by a variety of analysts. Other assessments are currently in progress. We describe the systems studied and then summarize the kind of problems found.

3.1 Systems Studied

Condor is a widely-used batch workload management system for high throughput computing. It provides a job queuing mechanism, scheduling policy, resource matching, data placement and staging, resource monitoring, resource management, and checkpointing. Its flexibility makes it the system-of-choice in a variety of situations, from being used to harness the CPU power of idle desktop or cluster nodes, to building Grid-style computing environments.

We have also applied FPVA to:

Storage Resource Broker (SRB) [2], a storage management system for Grid computing environments that can be incrementally deployed and provisioned. As is common with storage servers, SRB has a metadata service for naming, authentication, data location, and other attributes.

MyProxy [14], an X.509 public key infrastructure (PKI) credential management system developed at the National Center for Supercomputing Applications (NCSA). MyProxy combines an on-line credential repository with a certificate authority to allow users to store and manage, usually short lived, X.509 proxy certificates.

GLExec [8], an identity switching service that allows a Grid system to execute a remote user's job so that the job is isolated from the Grid middleware and from the jobs of other users. This isolation is accomplished by running the job with a user and group ID distinct from the middleware and other jobs. GLExec functionality is similar to Apache suEXEC [1] and was derived originally from that code.

Gratia Condor Probe [5], an Open Science Grid facility for robust, scalable, secure and dependable grid accounting services. The system consists of probes for acquiring accounting information about data transfers, storage allocation, site availability, and job accounting from remote locations in a network of one or more collector-reporting systems.

Condor Quill [9], an add-on to Condor developed at the University of Wisconsin that expands and improves the efficiency of data managing for the `condor_q` and `condor_history` query utilities by storing the required information in a central relational database.

CrossBroker [6], a resource management system for scheduling parallel and interactive applications.

3.2 Vulnerabilities Discovered

The systems studied and vulnerabilities discovered are summarized in Table 1. For each of the seven systems studied, we characterize the vulnerabilities by their impact. Most of the vulnerabilities are of a serious nature in that they allow an attacker to gain access to resources such as data or running code that should only be accessible to another account in the system or the underlying operating system. The impact is classified by the type of account that can be compromised in the order of lessening severity: root account (control everything on the host), admin account (control the system under study), and other account (access data of other unprivileged users). In cases where account privileges could not be elevated, the vulnerabilities all resulted in a denial of service (DOS) of the system.

The complete descriptions of the disclosed vulnerabilities can be found in the *Vulnerability Assessment* section of the MIST project web site [17].

The rest of this section presents a few types of vulnerabilities that were repeatedly found while assessing the projects.

The first type of vulnerability is a frequently-found structural flaw, namely failure to properly check the properties of a file name path, when the security of the application depends on the properties being true. An example of this can be found in CONDOR-2005-0004. In this vulnerability Condor reads the contents of a file that is writable by the Condor administrative user, and uses the contents of the file to determine executables that are started as root, but there is no check to make sure that only root could have created this file. This allows the Condor administrative user to elevate their privileges to the root account. This type of vulnerability can be discovered by looking at the file accessed by the system, how the contents are used, permissions and what types of constraints exist on the files properties.

A second type of vulnerability that we found repeatedly is the injection attack. Injections come in various forms and occur where multiple strings are combined, and processed by another component, but the next component does not parse the text as intended. It is easy to locate the site where some of these injections occur because common APIs are used to cause the text to be processed. In other cases, the site is not obvious. For instance, the `system` library function is a common cause of command injections. Determining if the call is actually exploitable can still be a difficult process as it involves determining if user controlled data of the right form can reach text passed to these APIs.

While scripting and SQL injections are well-known, we found a more exotic case where a Condor log file containing user supplied data was read by a Perl program that constructed a Python program that was then executed as root. In this example, there is a high level flow of data from the user to the Python script that is executed as root. The

Gratia Condor Probe, written in Perl, parses this Condor log file, and uses the user supplied data directly to create a Python string literal. A carefully crafted value can cause Python to parse the string as a string plus additional Python statements that are then executed. There is no facility in existing tools to track this complex flow of data though files and processes. Such a facility would likely improve the fidelity of the tool results.

These vulnerabilities are representative of the problems discovered. They include vulnerabilities that are implementation bugs that are localized in a small section of the code, and structural flaws that are not localized and require knowledge that cannot be deduced directly from the code alone. It should be possible for tools to discover many of the implementation bugs, but it is unlikely a tool will be able to discover structural flaws. The next section explores this further.

4. COMPARISON WITH AUTOMATIC TOOLS

Automated source code analysis tools are becoming increasingly important as a means to reduce the incidence of vulnerabilities in critical code. While these tools have a strong record in finding problems, our experience with in-depth analysis of middleware left us with two concerns. First, were the automated tools able to find the more complex and serious types of vulnerabilities? And, second, for mature software systems, would they be able to distinguish serious and exploitable problems from those that were less significant?

We studied the effectiveness of automated source code vulnerability assessment tools by comparing such tools to the results of applying our FPVA methodology to the Condor system. This study differs from previous ones in that it is not a comparison between automated tools; instead it compares the automated tools to a thorough FPVA study of a software system. The Condor vulnerabilities from the FPVA assessment provided a confirmed reference set of serious vulnerabilities. Such a study is important for understanding the limitations of the automated tools.

We found that while the automated tools are good at finding certain types of problems, they have some significant limitations. These limitations include reporting a large number of errors, most of which do not have security implications or are not exploitable (the false positive problem), and missing many vulnerabilities with serious security implications (the false negative problem).

A summary of the results of this study are presented in this section, while the full details of this study appear in [11].

We performed this study by applying two highly-regarded commercial tools, Coverity Prevent and Fortify Source Code Analyzer (SCA), to the Condor source code.

The most significant findings from our comparative study were: 1) of the 15 serious vulnerabilities found in using FPVA, Fortify found six and Coverity only one[1]. and all the vulnerabilities discovered by the tools were simple implementation type defects, 2) both tools 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), and 3) in the Fortify and Coverity results, we found no significant vulnerabilities beyond those identified by our FPVA study.

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. Given the finite resources in any assessment activity, these types of defects are rarely fixed. Ideally, a tool 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 system.

This comparison 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. Out of the remaining six vulnerabilities, Fortify did find them all, and Coverity found one. We expected a tool and even a simple tool to be able to discover these six 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.

5. RELATED WORK

Vulnerability assessment of software is an active field in both the research and commercial communities. In this section, we review a related methodology, vulnerability archive projects, and assessment tools.

5.1 Microsoft Methodology

The methodology that has the most in common with FPVA is Microsoft's threat modeling [10, 15]. It is aimed at identifying and rating the most likely threats affecting applications, based on understanding their architecture and implementation.

While Microsoft's methodology is the closest to ours, there is a key difference: after developing the architectural overview of the application, the Microsoft methodology applies a list of pre-defined and known possible threats, and tries to see if the application is vulnerable to these threats. As a consequence the effort is heavily biased towards finding known vulnerabilities, and the vulnerabilities detected may not refer to high value-assets. With FPVA, the component evaluation is performed only on the critical parts of the system, and we may be able to find vulnerabilities beyond a list, such as those resulting from the interaction of two components. In addition, under FPVA only threats that lead to an exploit are considered as vulnerabilities.

There is also a philosophical difference. While we advocated that FPVA should be performed by an assessment team independent from the developers, Microsoft suggests that the developers participate in threat identification. We believe that these interactions can lead to a biased analysis and may result in threats going undetected.

[1]note that we also have similar results from the use of veracode, ounce labs, and the most recent version of Fortify. the release of the full details of these results is awaiting government clearance, which should be available shortly.

5.2 Automated Analysis Tools

Automated analysis tools are an important part of the software development cycle. If they are applied to software starting in the initial coding stages, they can help to prevent many problems from appearing. Applying them later in the development cycle can be helpful, but has more challenges. These tools analyze the source code and report potential defects such as problematic uses of the language or APIs, programming style problems, potential buffer overflows, the use of tainted data in an insecure fashion, null pointer dereferences, and improperly acquiring and releasing of memory. As we discussed earlier, the best of these tools have significant limitations. We advocate continued comparison studies of these tools, especially with reference to their ability to detect known significant vulnerabilities and avoiding overwhelming the analyst with false positives.

6. CONCLUSION

Our work in vulnerability assessment has produced several key accomplishments and laid the foundation for our future efforts in securing Grid and Cloud computing environments.

Among our accomplishment are:

- *Development of the analyst-centric First Principles Vulnerability Assessment methodology:* FPVA has the important characteristic that it focuses on paths to the high value assets in a software system, and is not dependent on working from known vulnerabilities.

- *Applied FPVA to several key middleware systems:* These assessments include Condor, Storage Resource Broker, MyProxy, gLExec, Gratia, Quill, and CrossBroker, with ongoing studies of several more systems. These assessment identified many critical vulnerabilities, resulting in a direct improvement of the security of this software.

- *Increased the understanding of the effectiveness of automated code analysis tools:* Previous studies of such tools focuses on comparing such tools to each other, while not referencing a known set of serious vulnerabilities. Our study showed a major gap between what a trained analyst can do and what can be produced by the automated tools. These results provide a concrete direction for improving the capabilities of future automated tools.

Our ongoing work includes assessing additional Grid and Cloud systems, increasing the effectiveness of the FPVA analyst, and working to improve the automated assessment tools technology.

Acknowledgments

This research funded in part by Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL), NATO grant CLG 983049, National Science Foundation grant OCI-0844219, the National Science Foundation under contract with San Diego Supercomputing Center, National Science Foundation grants CNS-0627501 and CNS-0716460, and MEC-MICINN Spain under contract TIN2007-64974.

Our special thanks to TASC Inc. for their help in validating and extending our study of automated analysis tools.

7. REFERENCES

[1] Apache Software Foundation. *Apache suEXEC*. `http://httpd.apache.org/docs/1.3/suexec.html`.

[2] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, Toronto, Ontario, Canada, Nov.–Dec. 1998.

[3] R. A. Clarke and R. K. Knake. *Cyber War: The Next Threat to National Security and What to Do About It*. Ecco, First edition, 2010.

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

[5] Fermilab. *GRATIA, a resource accounting system for OSG, CD Document 2070-v1*.

[6] E. Fernández, E. Heymann, and M. A. Senar. Resource Management for Interactive Jobs in a Grid Environment. In *2006 IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 2006.

[7] Fortify Software, Inc. `http://www.fortify.com`.

[8] D. Groep1, O. Koeroo1, and G. Venekamp. gLExec: Gluing Grid Computing to the Unix World. In *International Conference on Computing in High Energy and Nuclear Physics (CHEP)*, volume 119 of *Journal of Physics: Conference Series*, Victoria, British Columbia, Canada, Sept. 2007. IOP Publishing Ltd.

[9] J. Huang, A. Kini, E. Paulson, C. Reilly, E. Robinson, S. Shankar, L. Shrinivas, D. Dewitt, and J. Naughton. An overview of Quill: A Passive Operational Data Logging System For Condor. Computer Sciences Technical Report, University of Wisconsin, May 2007. `http://www.cs.wisc.edu/condordb`.

[10] L. Kohnfelder and P. Garg. The Threats to Our Products. Microsoft Interface, Microsoft Corporation, Apr. 1999.

[11] J. A. Kupsch and B. P. Miller. Manual vs. Automated Vulnerability Assessment: A Case Study. In *First International Workshop on Managing Insider Security Threats (MIST)*, West Lafayette, IN, USA, June 2009.

[12] J. A. Kupsch, B. P. Miller, E. César, and E. Heymann. First Princples Vulnerability Assessment. MIST Project Technical Report, University of Wisconsin, Sept. 2009. `http://www.cs.wisc.edu/mist/papers/VA.pdf`.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor — A Hunter of Idle Workstations. *8th Intl Conf. on Distributed Computing Systems*, pages 104–111, June 1988.

[14] J. Novotny, S. Tuecke, and V. Welch. An Online Credential Repository for the Grid: MyProxy. In *Tenth International Symposium on High Performance Distributed Computing (HPDC)*, Redondo Beach, CA, USA, Aug. 2001.

[15] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.

[16] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency — Practice and Experience*, 17(2-4):323–356, 2005.

[17] University of Wisconsin. *MIST (Middleware Security and Testing) project*. `http://www.cs.wisc.edu/mist`.