

Implementing privilege separation in the Condor[®] system

Bruce Beckles

University of Cambridge Computing Service, New Museums Site, Pembroke Street,
Cambridge CB2 3QH, UK

Abstract

In this paper we discuss, in some depth, our restricted implementation of privilege separation for the Condor[®] system ([1], [2]) (in the Linux environment), and, in addition, we describe our proposed architecture for communication between privilege separated daemons in the Condor system. This architecture, if adopted, would allow each daemon to conform to principle of least privilege, thus significantly lowering the attack surface of the Condor system.

1. Introduction

The Condor[®] system ([1], [2]) is a batch scheduling system that is often used for “scavenging” the “idle time” on workstations whose primary purpose is not to run Condor jobs. It can also be used as a more conventional batch scheduling system, but its ability to “cycle scavenge” is considered to be one of its strengths. It is increasingly being deployed for this purpose across large numbers of workstations in UK academic institutions (for instance, at UCL [3], Cardiff University [4], the University of Cambridge [5], etc.).

The Condor project began in 1988 and Condor has been in continuous development ever since. Unfortunately, one of the consequences of this seems to be that Condor has not been designed with “security at its heart”, but rather security “features” have been added in a more “ad-hoc” fashion as it has developed.

As the modern network environment becomes an increasingly hostile place, it has become even more important that all network services are designed to be as secure as possible, even more so for systems as widely deployed as Condor. This is for a number of reasons, perhaps the most obvious of which is that it may be possible to gain unauthorised access to, or even control of, all the machines in an entire Condor pool simply by compromising a Condor daemon on an individual machine in the pool.

Thus the security of the entire Condor deployment may be crucially dependent on the security of the individual components of Condor on a single machine. This illustrates one of the central problems with the security architecture of the Condor system, namely that it has not been designed to enforce privilege separation [6] on its components or to follow the principle of least privilege [7].

In this paper we discuss how we have implemented a restricted form of privilege separation for Condor in the Linux environment. We also describe an architecture which, if implemented, would lead to a fully privilege separated Condor system, allowing the individual daemons of the Condor system to conform to the principle of least privilege and significantly lowering the “attack surface” of the Condor system.

The implementation of restricted privilege separation described here was developed for the Condor 6.6 series – the current stable release series – and has been tested to varying degrees with versions of Condor from Condor 6.6.5 to Condor 6.6.10. Note that in the following section we assume a certain degree of familiarity with the architecture of the Condor system. Those unfamiliar with Condor should consult [2] and [8].

2. Privilege Separation and the Principle of Least Privilege

Privilege separation [6] is the notion that those parts of a system that require different levels of privilege should be strongly isolated from each other. Properly implemented this helps to prevent privilege escalation [9], as even if one part of the system is compromised, those parts of the system with a higher level of privilege are not automatically compromised. The related principle of least privilege [7] states that a process should only ever have the lowest privilege it requires for its current task. If at some point it requires a higher privilege level for a particular task, it should only be granted that level of privilege at that point and immediately return to the lower privilege level when it completes the task in question.

In the current design of the Condor system, most of its daemons run with root privilege (the highest level of privilege on the system),

although they only need this level of privilege for a very small number of tasks. The Condor system has thus not been designed in accordance with the principle of least privilege. (Although note that it is possible to run the Condor daemons without root privilege at the cost of the system running insecurely, see [10].)

In addition, there is little privilege separation within the Condor system. For instance, those processes that listen to the network are normally running with root privilege – with the possible exceptions (see [11]) of the daemons that provide the functions of the central manager (the *condor_collector* and *condor_negotiator*) – although none of the network communication functions require this level of privilege. Were the components of the system properly privilege separated, it would not be necessary for any daemons that listen to the network to run with root privilege.

3. Design constraints of our implementation

In undertaking our implementation of restricted privilege separation in the Condor system, we operated under a number of design constraints, some of which had a significant impact upon our final implementation. Our principal constraints were as follows:

- We were not to make any changes to the Condor source code (although we had access to that source code). This was because we did not want to fork from the official Condor distribution and then have to maintain our own distribution.
- We were only *required* to implement privilege separation on execute nodes (i.e. machines which run Condor jobs). Achieving privilege separation for the other machine roles in the Condor system, whilst desirable, was not a priority. This was because we had tight control over the other machines in our Condor deployment (see [5] for details).
- We were only *required* to support Condor’s Vanilla universe [12] (support for other universes was desirable but not necessary). In fact, we were not required to support the more advanced features of this universe – the minimum functionality required was the ability to run jobs on execute nodes and retrieve the results. This was because this was the minimal functionality necessary to meet the essential requirements of our users.
- No daemons that listened to the network should run with root privilege. This is in

accordance with “best practice” for network services.

- Ideally, no Condor processes should run with root privilege. This would significantly reduce the attack surface of our Condor deployment.
- We needed to enhance our control of the job’s execution environment, as the features provided by Condor were not sufficient to allow us to meet the requirements of our Condor service (see [5]).

4. Overview of Condor job execution on execute nodes

Before examining the details of our implementation of privilege separation it will be useful to examine how Condor executes jobs on execute nodes. The following is an overview of job execution on execute nodes of jobs in the Vanilla universe (under Linux):

- The *condor_startd*, a Condor daemon that runs on the execute node and listens to the network, receives a request to execute a job.
- The *condor_startd* starts another daemon, the *condor_starter*, which retrieves the job’s executable and data files. These are placed in a subdirectory of Condor’s execute directory (which Condor considers to be the job’s working directory), with their ownership and permissions set appropriately.
- The *condor_starter* redirects standard error and standard output, and also sets the working directory, scheduling priority, certain resource limits and environment variables (including any environment variables sent by the user with the job) for the job. It then spawns, as a new process under a different user account, either the job’s executable or, if Condor’s *USER_JOB_WRAPPER* feature [13] is in use, the specified “wrapper” program (passing the name of the job’s executable and any passed parameters to it).
- If the job needs to be suspended or evicted for any reason, the *condor_starter* signals it appropriately.
- Upon job completion the *condor_starter* returns any files the job has left in its working directory, as well as the contents of the redirected standard output and standard error, and the job’s exit return code.

5. Privileges required by the Condor system

The Condor system consists of a number of interacting daemons. In order to implement privilege separation for this system it is necessary to first understand what privileges are used by each daemon. As our implementation of privilege separation is currently restricted to execute nodes, we will only discuss the `condor_starter` daemon, which runs only on execute nodes and, as described in Section 4, is responsible for starting and controlling jobs on the execute node. This daemon needs to be able to do the following:

- Start processes under another user account so that the user's job runs in a different user context to the Condor daemons. Under Linux, this requires the `CAP_SETUID` capability [14].
- Change the ownership of the job's executable and data files, and any files that the job produces during execution. This is so that the job can access its data and executables, and also so that Condor can return any files produced by the job and clean up after the job. Under Linux, this requires the `CAP_CHOWN` capability [14].
- Send signals to the job and any processes it spawns, so that it can suspend or evict the job, cause it to checkpoint, etc. Under Linux, this requires the `CAP_KILL` capability [14].

6. User context switching

From the overview of how Condor executes a job on an execute node (Section 4), it is clear that privileges normally granted only to the root account are necessary for at least some of the tasks it performs (see Section 5). Under UNIX and Linux these privileges are normally only available to processes running as root, which is why so many of the Condor daemons run with root privilege.

There are, however, ways of allowing a process that is not running as root to start another process in a different user context, e.g. the `su` and `sudo` commands, and `setuid` programs [15]. Unfortunately, there are problems with all these mechanisms when strong separation between the calling process and the called process is required. For instance, it is often possible for the calling process to set up environmental conditions that may have a malign effect on the called process.

Fortunately there are a number of other services or daemons, such as GNU `user` [16]

and superuser daemon [17], that allow one process to start another process in a different user context whilst enforcing strong separation between the calling process and the called process. We have chosen to use GNU `user` as it is already used elsewhere within our organisation, and so we have experience of using it in a production environment. In addition, by making use of a facility we are already using, as opposed to implementing or using a different one, we minimise the number of dependencies in our organisation's "chain of trust", which helps to keep our attack surface small.

There are a few features of GNU `user` that are particularly important for our implementation of privilege separation, namely:

- It allows arbitrary file descriptors to be connected across the security boundary between the calling and called processes; in particular, standard output and standard error.
- If the called process is successfully invoked, the exit return code of that process is returned to the calling process upon completion.

Those unfamiliar with GNU `user` are referred to [16] for further details, including an overview of how it works.

7. Overview of our implementation

We consider our implementation to be only a restricted implementation of privilege separation as we have not partitioned the functions of the Condor system into distinct processes based on their privilege requirements, as that would require extensive changes to the Condor source. However, we have gone some way towards ensuring compliance with the principle of least privilege, as the Condor daemons now run only with ordinary user privileges throughout their lifetime. Below is an outline of our implementation:

- GNU `user` is installed on all our execute nodes.
- No Condor daemon or process runs with root privilege, instead all the Condor daemons are started under a dedicated user account. (Note that when the Condor daemons are not run with root privilege they will not attempt to do any "privilege switching".)
- The Condor job and any processes spawned by it all run under a dedicated user account that is different to the account used for running the Condor daemons.

- Using Condor’s `USER_JOB_WRAPPER` feature, the `condor_starter` is instructed to pass all jobs to a wrapper script.
- This wrapper script immediately `exec()`’s another wrapper script with a “clean” environment.
- This new wrapper script now does the following:
 - Determines the scheduling priority and the resource limits set for the job by `condor_starter`,
 - Cleans up any files left by previous jobs,
 - Installs a signal handler to catch any signals from the `condor_starter` – if `condor_starter` attempts to signal the job, this will be caught and a `userv` service invoked that passes the signal on to the actual job process(es),
 - Kills any processes left behind by previous jobs (using a `userv` service),
 - Sets the appropriate permissions and ownership on the job’s executable, data files and working directory so that the job can access them (using a `userv` service where necessary),
 - Calls a `userv` service that starts the “job wrapper” script under the user account used for running Condor jobs, passing any necessary information,
 - Waits for this service to terminate, and
 - Upon termination of the service, stores the job’s exit return code, kills any job processes still running, changes the ownership on the files in the job’s working directory and any subdirectories, and exits with the job’s exit return code.
- The “job wrapper” script does the following:
 - Makes sure the correct resource limits are set for the job,
 - Sets the appropriate environment variables (retrieving and filtering any environment variables submitted with the Condor job), and
 - Finally, it `exec()`’s the job’s executable with the correct scheduling priority.
- The `userv` service that calls the “job wrapper” connects standard input, standard output and standard error across the security boundary, which means that the redirections put in place by the `condor_starter` daemon are respected. Thus Condor can still return the contents of the job’s standard output and standard error on job completion.
- Since neither the `SIGKILL` nor the `SIGSTOP` signals can be caught, these have to be handled differently:
 - Condor is configured never to try to suspend jobs (a restriction acceptable in our environment), so that the `condor_starter` will never send the `SIGSTOP` signal.
 - Using Condor’s `cron` facility [18], a script is run once a minute that, if no Condor job is running, kills any processes running under the dedicated user account used for running Condor jobs. Thus if Condor tries to send a `SIGKILL` to the job, it will terminate a wrapper script instead, and think that it has killed the job. Then, within a minute of this occurring, the actual job will be killed.

This implementation of restricted privilege separation does not currently support Condor’s Standard universe [12], at least in part because this uses file descriptors other than standard input, standard output and standard error. As GNU `userv` supports the connection of arbitrary file descriptors across the security boundary, that particular problem can be easily solved. However, there may be other issues that affect the functioning of the Standard universe and further investigation is required.

8. Benefits and limitations of our implementation

There are a number of benefits and limitations of this implementation. The principal ones are described below, followed by a discussing of how to address some of these limitations, etc.

8.1. Benefits

- No processes that listen to the network run with root privilege, bringing these components of the Condor system in line with accepted best practice for network services.
- No Condor daemons or processes run with root privilege, thus lowering the attack surface of our Condor deployment. Should the Condor system be compromised on any of our execute nodes, the entire operating system on that node will not automatically be compromised.
- We have much greater control of the job’s environment – if desired the `chroot` command could be used to run jobs (currently not supported by Condor) or jobs could be run in a separate “virtual machine” (e.g. User-mode Linux [19]).

- Our use of wrapper scripts means that we have a “hook” that allows us to run commands on *job completion*, a feature not currently available in Condor.
- Any processes left behind by the job after completion will be killed (Condor will normally only do this if specially configured), and in fact, such processes are checked for once a minute, which may provide some protection against processes that manage to “outrun” the initial SIGKILL signal.

8.2. Limitations

- This implementation only works for the Vanilla and Java universes [12], and not all the features of these universes are supported. In particular:
 - It is not currently possible to distinguish between a job that terminates by means of an unhandled signal and a job that terminates with a numerically high exit return code. This means that the ExitBySignal [20] and ExitSignal [21] ClassAd attributes are not supported.
 - Network socket inheritance across the security boundary is not implemented and so features such as Chirp I/O [22] (Java universe) may not function correctly.
- It is not possible to suspend jobs (as the SIGSTOP signal cannot be trapped).
- There is no support for Condor’s “virtual machines” [23]. (Note that these are not “virtual machines” in the sense that term is normally used but more like “virtual CPUs”, which allow more than one job to be executed simultaneously on a particular execute node. They are most often used to support machines with a symmetric multiprocessing (SMP) architecture.)
- When a Condor job is running, Condor can no longer correctly distinguish between the load on the machine due to Condor-related processes (which should include the Condor job’s processes) and the load due to other processes. This because it is unaware that the job is actually being executed under a different user account to that under which the Condor daemons are running. Thus it is not possible to enforce policies regarding Condor’s use of the machine based on the machine load. (Note that by default Condor uses such a policy.)
- Similarly, because Condor incorrectly thinks that one of our wrapper scripts is the actual

job process, the information it returns about CPU utilisation, the job image size, etc. is incorrect. Information about the job run-time, however, is reasonably accurate.

- When Condor attempts to kill a job there may be a delay of up to a minute before the job is actually killed (because the SIGKILL signal cannot be trapped, so we have to wait until our scheduled Condor *cron* job runs).
- The use of GSI authentication [24] for the Condor daemons is not supported as the libraries currently used by the Condor system to support GSI have – at least from the perspective of privilege separation – a flaw in their design. These libraries will refuse to allow so-called “server certificates” to be used for authentication if those certificates are not owned by the root account and accessible *only* to that account. This means that daemons using such certificates to prove their identity must have root privilege (or else they must have certain capabilities [14] normally only available to processes with root privilege).

8.3. Addressing some of these limitations

Many of the limitations described in Section 8.2 either can only be addressed, or are best addressed, by making changes to the Condor source code, and, as such, are probably best addressed by the Condor Team. However, further development of the implementation described here could address several of the others, most notably:

- Adding support for the Standard universe may well be straightforward, as indicated in Section 7, since the problems currently observed can be easily solved. However, as the Standard universe is one of Condor’s most complex universes, further investigation is required. As we are aware of very little use of the Standard universe within the University of Cambridge at present, adding support for this universe has not been a priority.
- To deal with the inability to trap the SIGSTOP and SIGKILL signals there are a number of options available:
 - Using Linux’s function interception facility through the LD_PRELOAD environment variable, it would be possible to write a library that intercepted Condor’s calls to the `kill()` system call and, when appropriate, used a `userv` service to signal processes instead. (This

would only work with dynamically linked versions of the Condor binaries.)

- By either taking advantage of the `ptrace()` function, or by monitoring the process table, it would be possible to determine when Condor sent a `SIGSTOP` or `SIGKILL` signal to our wrapper script, and to then take appropriate action.

Note that handling the `SIGKILL` signal would also solve the problem of the delay of up to a minute between Condor trying to kill a job and the job actually being killed.

- It is probable that the situation regarding jobs terminating due to an unhandled signal could be better handled as GNU `userv` offers a number of options for how it returns the exit return code of the process run by the `userv` service.
- Extending the existing scripts to handle Condor's "virtual machines" is fairly straightforward; the only reason that this has not been done is the lack of demand for this in our Condor deployment (where there are no execute nodes with SMP architectures).

The one limitation that cannot be addressed by changes to the Condor source code or by developing the implementation described here is the one to do with GSI authentication. Addressing this limitation requires changes to the libraries used to support GSI that were developed by the Globus Alliance (who are responsible for maintaining them).

8.4. Impact of these limitations in our environment

The Condor deployment for which this implementation of restricted privilege separation was developed ([5]) is one in which little of the advanced functionality of Condor is required, and so in which many of the limitations described in Section 8.2 are of little consequence. In addition, in our deployment, Kerberos [25] offers many advantages over GSI authentication (see [5] for a discussion), and there are no issues with using Kerberos with our implementation of restricted privilege separation. Thus it is certainly the case that, for our deployment, the benefits (Section 8.1) far outweigh the limitations (Section 8.2).

It is worth noting that in this deployment there is a very high degree of homogeneity of execute nodes, and that Condor jobs are evicted as soon as a user logs in to an execute node (whose main function is, in fact, to serve as a public access workstation). Further, the execute nodes are only made available to Condor during

periods when user activity is likely to be low (e.g. overnight). Thus much of the information about CPU utilisation, etc. – often used for meta-scheduling and for measuring efficiency – that would be returned by a standard Condor system is of relatively little value in this scenario.

8.5. Efficiency

One concern that is often raised about products whose components are privilege separated is that of efficiency. Additional overheads are likely to be introduced into a system by privilege separation. Partly this is because of the additional communication between privilege separated components that were previously a single component, and partly because of the overhead of making calls to the privileged part of the system (GNU `userv` in our case) in order to start processes in an different user context.

Notwithstanding such considerations, it is certainly possible to implement privilege separation with little impact on the performance of a system (see [6] for an analysis of the performance impact of privilege separation in OpenSSH). In our case, we estimate that privilege separation adds an additional 5 to 10 seconds at most – usually less – to job execution time. As our experience of the overheads of the Condor system itself are that they are such that it is not efficient to run jobs using Condor if the job execution time is 10 minutes or less, additional overheads of 5 to 10 seconds are unnoticeable.

9. A proposed privilege separated architecture for the Condor system

Although we have only implemented privilege separation on execute nodes, we have investigated implementing it on submit nodes (i.e. machines which submit jobs to the Condor system) as well. We concluded that it would be completely feasible to implement a similar form of restricted privilege separation *provided none of Condor's so-called "strong authentication" mechanisms were being used* (see [26] for details of Condor's authentication mechanisms).

The reason for this is that, in a privilege separated Condor system, the process on the submit node that manages an executing job (the `condor_shadow` daemon) would need to run as the user who submitted the job. However, if strong authentication were being used this daemon would need access to an authentication credential stored somewhere in the filesystem.

This would mean the credential would need to be accessible to all users who were allowed to submit jobs. In all but a few limited scenarios this means that the authentication credential is of no value for identification purposes, as it would be available to multiple individuals, some of whom may not be trusted.

We therefore propose the following architecture as an alternative (this architecture requires significant changes to the source code of the Condor system):

- GNU `userv` (or something with equivalent functionality) is used on all machines where the Condor system needs privileges not normally available to ordinary user accounts.
 - No Condor daemons or processes run with root privilege. (Note that this requires either that the libraries used for supporting GSI authentication are fixed so that they do not mandate that server certificates must be accessible only to the root account, or that GSI authentication is not used.)
 - All long running Condor daemons that listen to the network are split into separate components. One component (the network component) listens to the network and runs under a dedicated unprivileged account (distinct from the accounts used for any other components of the Condor system). This network component:
 - Runs in an environment in which `chroot()` has been used to change its root directory, and
 - Communicates with the other component(s) via some inter-process communication mechanism, such as IPC sockets – this allows, for instance, its logging functions to send their output to a file to which it cannot write via ordinary filesystem calls.
- In this discussion, a “long running” daemon is one that normally runs for all or most of the lifetime of the Condor system, such as the `condor_schedd` daemon, as opposed to one which is only started for a particular task and then exits, such as the `condor_shadow`.
- Condor job executables are run under an unprivileged account, different from any of the accounts under which any Condor processes are run.
 - Condor processes use appropriately constructed `userv` (or equivalent) services to start processes in different user contexts, to signal processes running in a different user context, and to change file ownership as necessary. Where necessary these services support passing file descriptors across the

security boundary between the calling process and the called process.

Handling authentication credentials for long running daemons is fairly straightforward: the network component simply passes the authentication messages back and forth between the network and the other component of the daemon via some inter-process communication mechanism such as IPC sockets, until a secure channel is established. For other daemons, such as the `condor_shadow` and the `condor_starter`, a number of other approaches are available, one of which is outlined here:

These daemons are typically run to handle one side of a transaction between two machines on the network. Thus it would often be possible to have their parent long running daemon negotiate a secure channel over a network socket on their behalf, which they then inherited when they were instantiated. Shortly before the validity of the secure channel was due to expire, they could use the session key established when the secure channel was created to “prove” their identity to a long running daemon (note that this is not a “strong” proof of identity). Having done so, they could then request re-negotiation of the secure channel and pass authentication messages back and forth between the long running daemon and the network using some inter-process communication mechanism (e.g. IPC sockets).

Note that the above is only an outline of our proposed privilege separated architecture and, as such, excludes all the fine details which are so important in ensuring the soundness of any security architecture.

10. Future work

The Condor Team have expressed interest in incorporating the work we have already done, or, more likely, equivalent functionality, in a release of Condor. In conjunction with them, we are currently investigating the best ways of doing this and the architectural issues around adding privilege separation to the Condor system. It is hoped that a preliminary form of privilege separation will appear in the next development release series of Condor, the Condor 6.9 series.

11. Conclusion

Although the Condor system in its current form was not designed with security as one of its central concerns, we have seen that it is possible to significantly lower its attack surface, at least on execute nodes, without making any

modifications to the Condor source code, at the cost of some functionality. This is possible by the implementation of a restricted form of privilege separation using a facility, such as GNU `userv`, that allows one process to call another process in a different user context. In our Condor deployment, the functionality lost is of comparatively little significance in comparison to the security and other benefits. For other deployments the situation may be different.

We have also formulated a fully privilege separated architecture for the Condor system, an overview of which is presented above (Section 9). This architecture would, if properly implemented, significantly lower the attack surface of the Condor system. However, architectures such as this one require changes to the Condor source code and so ideally should be undertaken by the Condor Team or in very close collaboration with them.

Acknowledgements

The author would like to thank the University of Cambridge Computing Service, particularly the Unix Systems Division, without whom this work would not have been possible. Thanks are also due to the Condor Team for their interest in, and support of, this work, and for their willingness to work with the University of Cambridge Computing Service and members of the UK e-Science community to address security issues in the Condor system.

References

- [1] The Condor[®] Project Homepage: <http://www.cs.wisc.edu/condor/>
- [2] What is Condor?: <http://www.cs.wisc.edu/condor/description.html>
- [3] UCL Research Computing Condor: <http://grid.ucl.ac.uk/Condor.html>
- [4] Condor – Cardiff University: <http://www.cardiff.ac.uk/schoolsanddivisions/divisions/instrv/forresearchers/condor/index.html>
- [5] Beckles, B. Building a secure Condor[®] pool in an open academic environment (2005). *Proceedings of the UK e-Science All Hands Meeting 2005*, Nottingham, UK, 19-22 September 2005, *forthcoming*.
- [6] Provos, N., Friedl, M. and Honeyman, P. Preventing Privilege Escalation (2003). *12th USENIX Security Symposium*, Washington, DC, USA, 2003: http://www.usenix.org/publications/library/proceedings/sec03/tech/provos_et_al.html
- [7] Saltzer, J.H. and Schroeder, M.D. The Protection of Information in Computer Systems (1975). *Proceedings of the IEEE* Volume 63, Issue 9 (September 1975), pp. 1278-1308: <http://web.mit.edu/Saltzer/www/publications/protection/index.html>
- [8] Condor[®] Version 6.6.10 Manual, Section 3.1: http://www.cs.wisc.edu/condor/manual/v6.6.10/3_1Introduction.html
- [9] McClure, S., Scambray, J. and Kurtz, G. *Hacking Exposed™: Network Security Secrets & Solutions*, Fifth Edition. McGraw-Hill/Osborne, 2005.
- [10] Condor[®] Version 6.6.10 Manual, Section 3.7.1.1: http://www.cs.wisc.edu/condor/manual/v6.6.10/3_7Security_In.html#SECTION00471100000000000000
- [11] Condor[®] Version 6.6.10 Manual, Section 3.7.2: http://www.cs.wisc.edu/condor/manual/v6.6.10/3_7Security_In.html#SECTION00472000000000000000
- [12] Condor[®] Version 6.6.10 Manual, Section 2.4.1: http://www.cs.wisc.edu/condor/manual/v6.6.10/2_4Roadmap_Running.html#SECTION00341000000000000000
- [13] Condor[®] Version 6.6.10 Manual, Section 3.3: http://www.cs.wisc.edu/condor/manual/v6.6.10/3_3Configuration.html#param:UserJobWrapper
- [14] capabilities (7) man page, Linux Programmer's Manual
- [15] Bishop, M. *How to Write a Setuid Program* (1986). *login: Volume 12 Number 1* (January/February 1986): <http://nob.cs.ucdavis.edu/~bishop/secprog/1987-sproglogin.pdf>
- [16] `userv` – user services client and daemon: <http://www.gnu.org/software/userv/>
- [17] `sudo` – superuser daemon: <http://sud.sourceforge.net/>
- [18] Condor[®] Version 6.6.10 Manual, Section 3.3: http://www.cs.wisc.edu/condor/manual/v6.6.10/3_3Configuration.html#8753
- [19] The User-mode Linux Kernel Home Page: <http://user-mode-linux.sourceforge.net/>
- [20] Condor[®] Version 6.6.10 Manual, Section 2.5.2.2: http://www.cs.wisc.edu/condor/manual/v6.6.10/2_5Submitting_Job.html#1843
- [21] Condor[®] Version 6.6.10 Manual, Section 2.5.2.2: http://www.cs.wisc.edu/condor/manual/v6.6.10/2_5Submitting_Job.html#1847
- [22] Condor[®] Version 6.6.10 Manual, Section 2.8.2: http://www.cs.wisc.edu/condor/manual/v6.6.10/2_8JavaApplications.html#SECTION00382000000000000000
- [23] Condor[®] Version 6.6.10 Manual, Section 3.10.6: http://www.cs.wisc.edu/condor/manual/v6.6.10/3_10Setting_Up.html#SECTION00410600000000000000
- [24] Foster, I., Kesselman, C., Tsudik, G. and Tuecke, S. A security architecture for computational grids (1998). *Proceedings of the 5th ACM conference on Computer and communications security*, San Francisco, CA, 1998, pp.83-92: <http://portal.acm.org/citation.cfm?id=288111>
- [25] Kerberos: The Network Authentication Protocol: <http://web.mit.edu/kerberos/www/>
- [26] Condor[®] Version 6.6.10 Manual, Section 3.7.4: http://www.cs.wisc.edu/condor/manual/v6.6.10/3_7Security_In.html#SECTION00474000000000000000