Evaluating Grid Portal Security

David Del Vecchio*, Victor Hazlewood** and Marty Humphrey*
*Department of Computer Science, University of Virginia, Charlottesville, VA 22904-4740
**San Diego Supercomputing Center, 9500 Gilman Dr, La Jolla, CA 92093-0505

Abstract— Grid portals are an increasingly popular mechanism for creating customizable, Web-based interfaces to Grid services and resources. Due to the powerful, general-purpose nature of Grid technology, the security of any portal or entry point to such resources cannot be taken lightly, particularly if the portal is running inside of the trusted perimeter, such as a Science Gateway running on an SDSC machine for access to the TeraGrid. To assess potential vulnerabilities of the current state of Grid portal security, we undertake a comparative analysis of the three most popular Grid portal frameworks that are being pursued as frontends to the TeraGrid: GridSphere, OGCE and Clarens. We explore general challenges that Grid portals face in the areas of authentication (including user identification), authorization, auditing (logging) and session management then contrast how the different Grid portal implementations address these challenges. We find that although most Grid portals address these security concerns to a certain extent, there is still room for improvement, particularly in the areas of secure default configurations and comprehensive logging and auditing support. We conclude with specific recommendations for designing, implementing and configuring secure Grid portals.

I. INTRODUCTION

According to the American Heritage Dictionary, a *portal* is "a doorway, entrance, or gate, especially one that is large and imposing". The intent behind such structures is really one of security, to allow the welcome visitors through, while keeping unwelcome intruders out. From a technological perspective, a portal is something that provides a convenient entry point to resources, applications or content located elsewhere. Early *Web portals* were typically web sites with search engines or indexes to other content on the World Wide Web [27]. Since all of the content accessible through these web portals was publicly available anyway, everyone was welcomed in and security was barely a concern.

In Grid computing, the resources of interest are not websites, but data and computational resources, services and applications. Thus the goal of a *Grid portal* is to provide a convenient entry point to these Grid resources, typically via a Web-based front-end. While many Grid portals expose relatively general purpose functionality like launching jobs for remote execution or retrieving remotely-stored data, they can also include application specific interfaces customized for a particular domain. Security gains prominence in Grid portals largely because of the nature of the Grid resources they expose. Many Grids link together powerful clusters of computational power and large scale data stores containing confidential, classified or proprietary information. A compromised Grid portal could allow an attacker to harness these powerful computational resources to launch a large scale attack elsewhere on the Internet or to gain user access to probe for privilege escalation or root compromise, for example.

Although Grid portals and Grid portal toolkits have existed since the early days of Grid computing, to date, to our knowledge no comprehensive analysis exists regarding the security and/or potential vulnerabilities of Grid portals. The analysis reported in this paper is based on our experience designing and implementing secure Grid software (Legion, WSRF.NET) as well as our experience securing a large supercomputing center (SDSC). We are particularly driven by the use-case of the TeraGrid Science Gateways. Before a TeraGrid site will run such a Grid portal *inside* their trusted perimeter, and particularly given recent increased Federal requirements regarding national computing infrastructure (e.g., FISMA [8]), it is crucial that Grid portal technology be assessed with regard to security, and best practices for running Grid portals be established.

In order to assess the current state of security with respect to Grid portal technology, we analyze the three popular portal implementations. Clarens [20] is a framework for writing Grid applications motivated by the Compact Muon Solenoid (CMS) experiment and processing application. Originally developed for Apache/mod-python, there is also a Java implementation [2] now available. GridSphere [17] is a generic Java-based framework for writing web and Grid portals through the standardized Java portlet API [1]. Also included are a set of Grid portlets for proxy credential retrieval, file management and job submission. The Open Grid Computing Environment (OGCE) [3] is a set of Java portlets and libraries for various Grid computing tasks that can run in either the GridSphere or uPortal [24] containers.

After reviewing some basic security requirements and potential vulnerabilities of portal technology (Section 2), we analyze these three Grid portal implementations. First, we present the overall design and architecture of the portals (Section 3) then examine how they address Grid security needs, specifically in terms of authentication, authorization and auditing (Section 4). We find that although most Grid portals address these security concerns to a certain extent, there is still room for improvement, particularly in the areas of secure default configurations and comprehensive logging and auditing support. We conclude with recommendations for securely deploying, configuring, running and maintaining Grid portals (Section 5).

II. Security Requirements and Vulnerabilities

Most Grid portals are just Web applications [26] that provide a front-end interface to accessing various grid resources. Although the architecture of Web applications vary, many applications leverage similar sets of software components. The **Web Server** (examples: Apache, IIS) processes incoming web requests, often routing them to other pieces of software. This is the outer layer of software for most Web applications, serving up static content when requested or delivering content dynamically generated by an individual application. The **Web Application Container** (examples: ASP/ASP.NET and J2EE/servlet containers like Tomcat, JBoss, Weblogic, Websphere) is a layer of software/libraries to facilitate writing, deploying and running Web applications. This typically abstracts away details like communicating with the Web server and process lifetime management so Web application authors can focus on exposing useful functionality through dynamically-generated content. Other technologies like CGI and PHP (which runs as a module in the web server) provide some similar capabilities, but don't as strongly present this notion of container. The **Web Applications themselves** (e.g., Grid portals) usually runs in a Web Application Container and often communicate to some backend Data Layer when generating content. Finally, the **Data Layer** is usually a back-end database, but could also be a file system or other storage mechanism.

A. Web Application Security Requirements

In the field of information security, three core principles are confidentiality, integrity and availability. Many experts also add accountability to the list as a fourth component [10]. A variety of mechanisms can be employed to achieve these three (or four) elements of information security, but many systems (at minimum) rely on a combination of mechanisms from the following broad categories.

Authentication is "the process of verifying an identity claimed by or for a system entity." [19] Authentication requirements can be divided into *Grid portal account creation* and subsequent *run-time authentication*, as when the authorized user attempts to use the portal for job submission or data access. Some Grid portals are architected to multiplex all users onto a single Grid account/ID (e.g., "CMS"), while other portals are predicated on the portal user *already* having a Grid account (and the goal of the Grid portal in this case is to perform Grid computations on a per-user basis as that particular user). In both cases, most web portals and applications will require user identification at portal registration time and authentication of user identities to (1) limit system access (authorization) based on this identity and to (2) tie a record of system actions to this identity (for auditing).

When only a single account/ID exists for the entire portal, the Grid portal requirements for user registration and identification are stringent. Section IA-4 from the FISMA regulation [8] is representative of such guidelines, stating that an organization (in this case the people who have control over the Grid portal) should manage user identifiers by (i) uniquely identifying each user; (ii) verifying the identity of each user; (iii) receiving authorization to issue a user identifier from an appropriate organizational official.; (iv) ensuring that the user identifier is issues to the intended party; (v) disabling the user identifier after [organizational defined time period] of inactivity; and (iv) archiving user identifiers.

When jobs and/or data access via the Grid portal take place based on a per-user pre-existing account, account creation is not as critical, because presumably an attacker could not launch jobs just by having a Grid portal account. Nevertheless, the guidelines of FISMA should be followed, albeit with potentially less dire consequences if such procedures are not properly implemented.

After the account has been created on the Grid portal, there are a number of alternatives for authenticating to the Grid portal to access the Grid for either job execution or data access. The HTTP protocol includes *Basic and Digest authentication*, which is built into most web browsers and web servers. Since these are basically plaintext protocols, Basic and Digest authentication should generally be avoided especially for password-based security mechanisms.

Another very popular approach is *forms-based authentication* in which the security token (often a password) is entered in a web form for input to the web application. Forms-based authentication gives a great deal of flexibility and control to the web application designer, but because of the plaintext nature of HTTP, care must be taken to protect the submission of the security token (usually SSL). Because of the relatively unconstrained nature of form input data, careful input checking must be done to avoid security vulnerabilities. A common problem with password-based authentication is that user-chosen passwords are often relatively weak and easy to crack.

In contrast, c*ertificate-based authentication*, most often via SSL is often considered a stronger way to establish the client's identity. Other strong forms of authentication including smart cards and biometrics exist as well, but SSL has the advantage of being built into most web servers and browsers. Among the challenges with PKI-based X.509 certificates is that the user's certificate must usually be physically located on a particular machine and establishing proper trust hierarchies through various different certificate authorities (CAs) can difficult to setup and get right.

In most cases the web application with be interested in authenticating its users, but it can also be important for users to verify the identity of the web application. The most common way to do this is via the certificate based server authentication of SSL/HTTPS. Many attackers exploit users' confusion about the web application's identity to trick them into revealing sensitive information.

Authorization is the process by which a "right or a permission is granted to a system entity to access a system resource." [19] The goals of authorization or access control are to restrict system access to those users or entities which actually need it. In addition to preventing unwelcome, unwanted system usage, access control is also used to limit the possible actions that are allowed by a properly authenticated user. Typically, this means following the principle of least privilege: a user should only be granted the lowest authorization level needed to carry out the desired action. Giving all users administrative-level access may be an easy way to ensure that valid user requests aren't rejected, but it's hardly a secure configuration.

Authorization is most commonly managed through *access control lists* (user-centric) or *capabilities* (resource-centric). Simple lists can be fine when the set of users and resources is small, but gets to be difficult to manage as these sets grow. One way to combat this complexity is *role-based access control*, in which users are assigned roles and these roles are used as the unit of authorization. Another variation is to assign attributes to the users and then restrict access to resources based on the attributes presented. Access to a resource is usually not all-or-nothing, and authorization incorporates what action or operation is intended as well.

Important considerations for web application authorization involve out-of-the box (default) security and ease of proper configuration. The default configuration for a newly-installed web application should be very restrictive, but often this is not the case. Many default configurations allow access to everyone, or include default guest or administrator accounts with well known passwords. Properly configuring access control mechanisms can also be challenging for web applications. A highly sophisticated, securely programmed authorization mechanism is worthless if it is not configured correctly so care must be taken to enable administrators to easily allow access only to those that need it.

Auditing is the process of verifying that security requirements have been satisfied, with corrections suggested where they haven't been met. Essential to effective auditing is that actions are traced and logged through all parts of the system. With web applications, this means that logging of significant operations must happen in the web server, web application and data layer, in addition to the web application itself. Events of interest include: errors, failures, state accesses, authentication, access control and other security checks, in addition to application-specific operations and actions. Care must be taken to protect the integrity of logging and trace data, even (and perhaps especially) in the case of system failures. Logs that are tampered with or destroyed are useless in performing an effective audit.

Auditing of log and trace data can either be done manually or it can be automated and often a combination of both is used. Either way, auditing should be done on a regular basis. Automated systems that continually monitor, detect, and in some cases even correct (or at least recommend corrections for) security problems can be particularly useful for maintaining a secure web application. Somewhat related to logging and auditing, Web applications should be careful to ensure that errors or failures somewhere in the system do not introduce security vulnerabilities. Attackers, for instance, are often able to exploit the detailed error information provided by web applications to gain unauthorized access.

Session Management. Web-based applications, in contrast to desktop-based application clients, have a challenge with regard to where client-related state information is stored. A desktop application would store state locally on the client machine, but because of the relatively stateless nature of the web browser, client state in web applications tends to be stored remotely on the server. The challenge then becomes securely managing and associating session state with an authenticated client identity. Unlike the other areas mentioned above, session state management is not strictly a security concern. However, the potential for security vulnerabilities in this area as well as its unique relevance to web applications merit its discussion here.

Many web application containers include built-in session management capabilities, and in most cases it is desirable to leverage this functionality where possible. When session state management must be built by the web application, care must be taken to ensure that session state can not be tampered with and is securely (i.e., cryptographically) and consistently mapped to an authentication token. From the client perspective, session identifiers are often included in cookies that are automatically saved and presented by the web browser. Such information could also be presented elsewhere in user input data. Care must be taken to protect the integrity and confidentiality of these session identifiers as attackers can use this information to gain unauthorized access to the system (see the attack scenario below). As much as possible web application interfaces should be constructed such that users can keep their session state secure (often this means including sensible logout procedures, among other things).

B. Grid Portal Security Requirements

Generally, the security needs of web applications discussed above apply wholesale to Grid portals as well. To their advantage, Grid resources tend to have their own security (authentication and authorization in particular) mechanisms in place, so breaking into a Grid portal, while concerning, may not necessarily allow the attacker access to backend Grid resources. For instance, simply being able to submit jobs through a portal is not useful without proper Grid credentials to authenticate to the actual job execution service. Consequently, the key security challenge of most Grid portals is that at some level, *they manage Grid credentials on behalf of clients.* Compromised Grid credentials are an extremely serious security breach because they allow an attacker to effectively impersonate a valid Grid user until the credentials are revoked or expire.

Thus, extra care must be taken in the management of these Grid credentials, which can effectively be viewed as a special kind of session state. The integrity and confidentiality of these credentials must be maintained even in the case of errors or failures. Accesses to the credentials should be logged and monitored continuously for suspicious behavior. Further the credentials, especially if stored on disk must be protected from other users or applications running on the web application server. A compromise elsewhere in the server's software stack should not lead to compromise of user's Grid credentials.

C. Vulnerabilities of Web Applications

A great challenge in developing secure web applications is that the vulnerabilities in any component of the architecture can often result in compromise of the web application as a whole. For instance, even though the code of a particular web application might be carefully written and free of security holes, vulnerabilities in the web server could still be exploited, causing the secure web application to be hijacked or overridden with a malicious version. Another challenge of the architectural complexity of many web applications is that it is often difficult to configure all of the components correctly and securely. So, even if the components as developed are free of security vulnerabilities, misconfiguration can unwittingly open the web application to compromise.

The Open Web Application Security Project (OWASP) compiled a list of ten of the most common security vulnerabilities afflicting Web applications [22] (and thus Grid portals, which are just a specific type of web application):

- **Unvalidated Parameters** – input contained in web requests is not properly checked (by the application) before being acted on. Attackers can craft parameters to hijack the application or cause it to behave in dangerous, unexpected ways. *Injection Flaws*, *Buffer Overflows*, and *XSS Flaws* are all specific types of *Unvalidated Parameter* vulnerabilities.
- **Broken Access Control** – access control mechanisms work inconsistently or incorrectly, allowing unintended access to resources. This is particularly troublesome for web application administrative interfaces.
- **Broken Authentication and Session Management** – authentication problems can range from weak authentication mechanisms that are easily broken (plain text secrets to retrieve forgotten passwords), to insufficient session protection (exploiting access to one set of session information to gain access to someone else's) to forged sessions or session cookies (allowing session impersonation).
- **Cross-Site Scripting (XSS) Flaws** – involves exploiting an unvalidated parameter vulnerability to send a script to the web application that is in turn delivered to and executed by the end user's web browser.

- **Buffer Overflows** – specially crafted input results in the execution of arbitrary code on the target server. This is particularly problematic if the server is running as root or an administrator account as the malicious code will also have those privileges. In general, Java applications do not suffer from this type of vulnerability (although the JVM itself could).
- **Injection Flaws** – in contrast to the other unvalidated parameter attacks, this refers to when injected code or command strings are passed through the web application directly to some backend system. SQL injection attacks are probably the most common.
- **Improper Error Handling** – this type of vulnerability surfaces when error messages displayed to the user in some way reveal details about how the system or application works (the attacker could then exploit this knowledge). This is typically a problem when very detailed stack traces are displayed to the user giving some information of the structure of the code and its operation. Attackers can also probe for inconsistencies in error messages returned ("file not found" vs. "access denied") to gain a better understanding of the application.
- **Insecure Storage** – storage of sensitive data (passwords, account information, etc.) without proper encryption or access control mechanisms. This could be on disk, in a database, or in memory. Usually one of the other exploits is needed to actually gain access to this insecure data.
- **Denial of Service** – when the sheer volume of requests to the web application overwhelms the capacity, denying access to legitimate users. This is usually an even more troublesome problem as web server DoS attacks (like SYN flooding), because it's very hard for web applications to distinguish between legitimate and malicious requests. The complexity of web applications usually means a fairly low threshold of concurrent connections needs to be exceeded to deny access.
- **Insecure Configuration Management** – problems here range from unpatched software to unchanged insecure default settings to outright configuration mistakes caused by incomplete or incorrect understanding of some very complex software. Clearly this is a human problem as much as a software problem, but delivering software that's easy to understand, easy to configure and comes in a secure configuration out of the box would certainly help.

III. Architecture Of Grid Portals: GridSphere, OGCE, and Clarens

To get a better appreciation for the specific security requirements and features of the three Grid portals in our comparison, we first detail their design and architecture. Both GridSphere and OGCE rely on the Java portlet specification, JSR-168 and as such their architectures are similar because of the constraints of this specification. The OGCE portlets, being standards-compliant, can theoretically run in a number of different portlet containers, including GridSphere. To broaden our discussion, we will assume that OGCE is running over the uPortal portlet container instead. When it comes to specifics, we distinguish whether a design or feature is a function of the portlet container or elsewhere in the software. Clarens also has two implementation choices: Apache/mod_python and Java (JClarens). For the purposes of our discussion here, only the Python implementation is considered, although bear in mind that the server-side APIs are similar between the two implementations and the web client support is targetable to either service implementation.

A. GridSphere

GridSphere consists of a JSR-168 [1] compliant portlet container along with a collection of general-purpose utility and Grid-specific portlets. As defined by the specification, a *portlet* is a Java web component that generates dynamic content in response to processed requests. Portlets are managed by a *portlet container*, which in addition to providing a runtime environment to the portlets, manages their lifecycles and provides them with a persistent storage mechanism. The portlet container specification effectively extends the Java Servlet specification [7] and portlet containers are expected to support the functionality described by this latter specification as well. GridSphere relies on the Apache Tomcat [5] servlet container to host the GridSphere portlet container (see Figure 1).

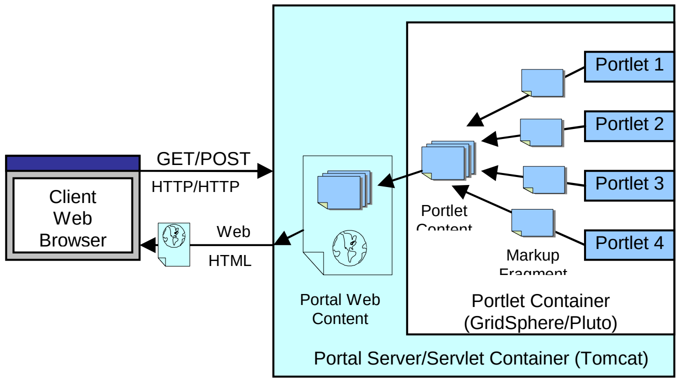

Figure 1: Architecture of a Java Portlet-based Portal

A portlet is typically displayed as a small window inside of a web page (complete with minimize-maximize and close buttons). Thus a user interface in portlet containers (including GridSphere) is constructed from one or more of these portlet windows, often grouped into tabs (see Figure 2). The portlet lifecycle is managed by the portlet container, which drops into user-developed code at well-defined times:

- when the portlet is loaded – *init()*
- when the portlet's interface is rendered – *render()*
- when the portlet's user interface is manipulated, i.e. an action or event
- when the portlet is unloaded – *destroy()*

GridSphere also includes a manager portlet to allow for dynamic loading/re-loading of portlet classes. Portlets much like servlets are packaged in to a Web ARchive (WAR file, similar to a JAR file) and are configured via an included deployment descriptor file. GridSphere includes a custom user-interface tag library for Java Server Pages (JSP) to hide browser-specific HTML and enable a consistent look and feel across portlets (JSPs are the standard way to construct portlet user interfaces). GridSphere relies on Hibernate [9], an object-relational persistence service, to hide the underlying details of the specific database used and allow developers to access persistent data through Java objects instead. Hibernate works with most databases that have a JDBC driver including MySQL, DB2, MySQL and others.

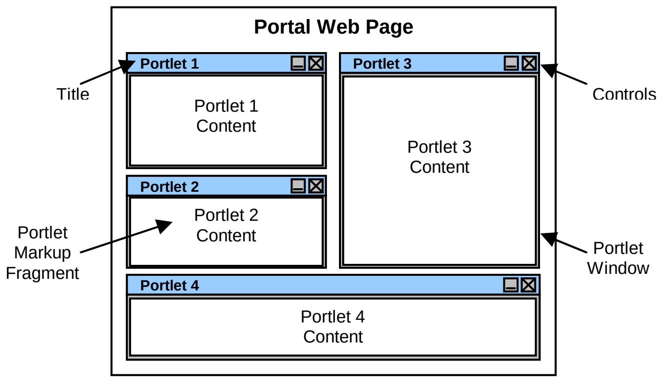

Figure 2: User interface of a portlet-based web portal. Figure adapted from the Java Portlet Specification [1].

In addition to the portlet container, the GridSphere distribution includes a collection of basic user/administrative portlets and some Grid-specific portlets. Portlets are provided for users to login/logout, manage profile information, adjust portlet layout/placement and modify the set of portlets they are currently subscribed to. Administrative portlets are provided for managing the set of users and groups the portal recognizes and for starting, stopping or redeploying portlets as needed. Grid-specific functionality in GridSphere is structured as re-usable "services" that can be shared across several portlets. These services include well-defined APIs for portlet developers and make use of the Java CoG [25] to actually carry out Grid-related operations. The included services are:

- *CredentialManagerService*, for accessing (listing, adding to, removing from) the set of credentials currently available for a given portal user
- *CredentialRetrievalService*, for retrieving a credential from a credential repository such as MyProxy
- *FileBrowserService*, for accessing remote file resources (filesystems) and performing basic file operations (list, change directory, etc.)
- *LogicalFileBrowserService*, similar to the FileBrowser, but for logical rather than physical file resources
- *JobSubmissionService*, for creating job specifications and submitting them to job resources
- *ResourceRegistryService*, for discovering, listing, adding or removing any of a number of Grid-related resources of interest

Developers are certainly free to make use of these services when writing their own application specific portlets, although GridSphere does distribute a collection of their own Grid Portlets for: credential management, resource browsing, job submission and file management.

B. OGCE

The Open Grid Computing Environments Collaboratory (OGCE) has developed its own set of JSR-168 compliant portlets for Grid portals. We will consider the combination of these portlets deployed in the uPortal portlet container. Unsurprisingly, the architecture of this combination is quite similar to that of GridSphere/Grid portlets with much of the design dictated by the Java Portlet Specification (Figures 1 and 2 are applicable). The goal of uPortal is to be a framework for building portals to serve the members of a university community. As much uPortal development occurred before the portlet specification was finished. uPortal initially developed its own portlet-like concept called a channel. Recent work has transitioned uPortal to more complete compatibility with the portlet specification, with portlets as the natively supported unit of portal content. uPortal relies on Apache Pluto [4] the reference portlet container implementation for this support. uPortal, like GridSphere uses Tomcat, another Apache project, as its hosting servlet container.

Regarding user interface and presentation, a pair of XSLT stylesheets is used to transform a user layout XML document into the desired page structure and theme. For persistence, uPortal relies on the Spring JDBC [13] library which is similar to vanilla JDBC, but with some enhancements making connections easier to manage and query results easier to translate into business-level objects (short of full object-relational mapping). Internally, the uPortal framework makes use of several classes that manage various aspects of the portal's operation including: SessionManager handling http/servlet sessions, UserLayoutManager handling portlet layout and preferences, ChannelManager for managing portlet instances, ChannelRenderer for handling portlet rendering/refresh and a PropertiesManager for aggregating portal configuration and making it available to other classes.

Although uPortal does include some sample portlets/channels, the most important security implications are from the Grid-related functionality exposed by the OGCE portlets. Four core portlets are currently included with OGCE, most of which make use of the Java CoG under the hood to abstract the underlying Grid protocols.

- Proxy Manager Portlet, for retrieving credentials from MyProxy servers
- File Manager Portlet, using GridFTP to browse the contents of remote file system and upload/download files to/from the desktop (or 3rd party transfer between machines)
- Job Submission Portlet, using GRAM to submit jobs for execution remotely
- Information Services Portlet, using GPIR [18] to display the current features and status of various storage and compute resources within or across organizations

C. Clarens

Clarens differs from GridSphere and OGCE in that its goal is to enable development of both client and server-side in a generic way. Services in Clarens are usually XML-RPC or SOAP-speaking Web services, and clients can be any Web service clients. Clarens clients are not constrained to the Web browser, although it is a supported interface option and perhaps the one of most interest here. The Clarens server is implemented in Python and server code gets triggered in response to Web requests to an Apache web server through the mod_python module [16] (see Figure 3). Clarens will look at the HTTP headers, URL and content of the request to determine how to process it: GET requests return the requested file (if it exists), while POST requests result in XML-RPC or SOAP responses as appropriate. Clarens handles the serialization/de-serialization of these messages, freeing service developers to concentrate on developing service logic in Python. Core functionality in the Clarens server includes the following:

- Security (authentication and authorization)
- Service Discovery
- Session Management (persistent data storage)
- Logging
- Plug-in Management

For storing session data, Clarens uses a database called TDB [21], which has built-in support for multiple simultaneous writers and internal locking. New services in Clarens can be deployed simply by placing the Python files in the proper directories and setting up a configuration-file with service-specific parameters in the configuration directory.

There are a number of Grid-related service implementations distributed with the Clarens server, including:

- Proxy escrow service, for managing Grid credentials
- File management service, for listing directory contents, file read and write
- Shell service, for executing arbitrary commands via a remote shell-like access
- Registry and discovery service, for tracking resources and availability

There also exist some management services for grouping services into an organizational hierarchy and managing access control for various aspects of the server and its services.

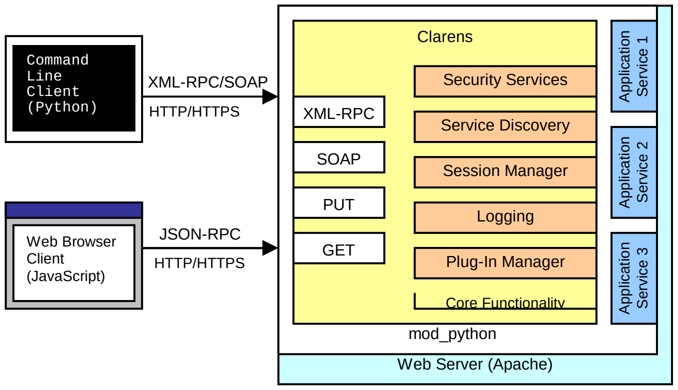

Figure 3: Clarens Architecture

Clarens clients could theoretically be developed in any language with Web services support, but in practice, most developers will want to take advantage of the

Clarens-provided client library support. Library support for message security and XML serialization makes Python-based command line clients relatively straightforward to author. Browser-based Web clients are the other primary interface option, with Clarens relying on JavaScript and JSON-RPC for this support. The JavaScript Object Notation (JSON) [11] is a simple, compact data format based on the JavaScript language. JSON-RPC [14] leverages this data format to encode simple, two-way remote procedure calls over a connection (often a TCP socket or HTTP). The interface itself is constructed as a combination of HTML for input, formatting and display and JavaScript for submitting the JSON-RPC request to the Clarens server and processing the response. A Python implementation of JSON-RPC is used to process these requests in the Clarens server.

IV. Grid Portal Security Features

Although there are many facets to portal security, in this section we will primarily compare the security features of the different portal implementations in four key areas: authentication, authorization, auditing and session management. Section 5 includes some discussion of other areas of security along with recommendations. Throughout the discussion in this section, we assume communications with backend Grid resources follow properly secured protocols (e.g., GSI/TLS), and instead we focus on interaction between the client software and the Grid portal server and subsequent vulnerabilities.

A. Authentication

GridSphere supports authentication via one or more configurable modules, similar to the Pluggable Authentication Modules (PAM) support of UNIX systems. The default module authenticates based on a username and password (hashed) stored in the GridSphere database. From a client perspective, this would appear as username and password text fields in the portal entry page or forms-based authentication as discussed in Section 2.2. Another included authentication module is an adapter for authentication via the Java Authentication and Authorization Service (JAAS) API [12]. A number of JAAS LoginModules exist that include support for LDAP-based authentication, among other things. If more than one authentication module is configured, GridSphere will try them all, in configured priority order until one succeeds.

The authentication features of uPortal are similar to GridSphere in that it supports several different authentication options, depending on configuration. The default configuration includes a login channel (or portlet) with a form for username/password entry. This form data is submitted to an authentication servlet which in turn engages an authentication service. The authentication service is responsible for performing the actual authentication, looking up user attributes (via a directory service such as LDAP) and setting up the user’s context and layout based upon the authenticated identity. Authentication modules must conform to a uPortal-defined Security Provider interface, with the default module doing a simple hash-based username/password lookup in the portal’s database. Once authenticated, an interface is provided to make user principal information, attributes and preferences available across different portlets.

Authentication to Clarens services is primarily X.509 certificate-based, over the SSL handshake protocol. This could either mean mutual authentication at the transport level (i.e., HTTPS) or Clarens also supports a packaging of the SSL handshake data in the AUTHORIZATION header as for HTTP Basic authentication. A configurable CA certificates directory is used to setup the trusted root certificates on the server and Clarens also supports the use of proxy certificates [23] for client authentication. The Clarens Web-based interface typically follows the same authentication approach, but also enables a password-based option for users that may not have their certificate installed in the browser of a particular machine. A login form allows users to specify a DN and password for a previously uploaded proxy certificate to authenticate to the Clarens server and allow it to lookup credentials to use for subsequent interactions.

Comparing the various portal authentication strategies we’ve just discussed, we see that GridSphere and uPortal both use some kind of forms-based authentication by default and Clarens also supports this as a secondary authentication option. Forms-based authentication (See Section 2) necessitates meticulous input checking to avoid security vulnerabilities and usually requires encrypted transmission as well. Although all three portals could theoretically be run over HTTPS, none of the portals require such a configuration and all of them expose HTTP non-encrypted communications in their default configurations. Clarens would seem to have an edge in its built-in support for SSL-based mutual authentication. While there don’t seem to be obvious security problems with their handshake-over-HTTP Basic authentication protocol and implementation, widely distributed, well tested security libraries are usually preferred over custom implementations (i.e., HTTPS is the safer bet). Clarens also lacks support for pluggable, custom authentication modules, both a curse and a blessing. A curse because it limits authentication flexibility; a blessing because poorly written custom authentication modules are an easy way to introduce security vulnerabilities.

B. *Authorization*

Portlets in GridSphere are assigned to groups either statically or dynamically and each group has its own access control settings. As such, it implements a form of *role-based access control*. Portlet groups can be designated: *public*, meaning that any authenticated user has access; *private*, meaning that users are aware of the group's existence but require an administrator to grant access; or *hidden*, meaning the group is private and hidden from users that don't have access to it. Granting users access to portlet groups is primarily done through a Group Manager Portlet that only administrators have access to. The other dimension of authorization in GridSphere is user roles. Currently users can be assigned to one of four role categories for the portlet groups they have access to: *guest*, *user*, *administrator* or *super-user*, each with their own privilege levels. Interestingly, the default configuration for GridSphere is to allow anyone who visits the portal login screen to create a user-level account for themselves (although this capability can be restricted to administrators only). From a developer's perspective, this authorization functionality is built into a "portlet service," which is basically some encapsulated software functionality that can be shared across multiple portlets. Portlet authors could certainly leverage the standard AccessControlManagerService (whose behavior was just described) included with GridSphere, or could theoretically write their own authorization portlet service. However, as of this writing, limited documentation on the latter could make for a risky undertaking from a security perspective.

The fundamental units of authorization in uPortal are called *Permissions*. Permissions are granted by portlet owners to user *Principals*, giving rights to perform certain *activities*. Permissions can also specify a timeframe in which they are valid and a target resource to which they apply. So, an example permission might grant user Bob (*principal*) the right to subscribe (*activity*) to the job submission channel/portlet (*target*). Note that in addition to individual users, a principal can also refer to configurable groups of users, simplifying administration. Every portlet and channel each has access to its own *PermissionManager,* which can store and retrieve permissions related to the target, enabling it to answer access control questions for the permission owner. Principal groups can also have permissions associated with them; these would deal mostly with group management (who can add users to a group, etc.). An *AuthorizationPrincipal* functions as the permissions manager for a group; this is primarily an access control list model rather than a capabilities model. From an administration perspective, uPortal includes two management interfaces, a ChannelManager and GroupsManager that can be used to configure permissions for channels/portlets and principal groups, respectively. The use of abstract interfaces to represent these permissions management classes suggests that it might be possible to insert customized authorization controls. How one might go about this, however, is somewhat unclear. The next major release version of uPortal aims for more sophisticated pluggable authorization architecture.

Clarens relies on a collection of access control lists as its authorization mechanism. An access control file can specify who is permitted or denied access to a particular service module as a whole or to specific methods that module exposes. Access control for files is similar and controllable on a per-directory or per-file basis. Users in Clarens are identified by their distinguished names (DNs) and these DNs can be assigned to hierarchical groups as desired. Members of higher-level groups are automatically members of groups below them in their branch of the tree and a Clarens admins group is at the root of the hierarchy. It is either these groups or individual DNs that are then used to define access control policy for methods and files. Through the Clarens web interface, it is possible to manage groups and edit access control lists as an alternative to editing these configuration files by hand.

All three portal frameworks rely on some form of access control list-based authorization, but vary in their level of sophistication and embellishment. Clarens has the simplest mechanism overall; easy understood, it may satisfy most authorization needs. Unfortunately, the Clarens web based ACL configuration interface doesn't add much in the way of ease of use and may actually be more confusing than hand-editing the configuration files. uPortal is at the opposite end of the complexity spectrum with support for customizing the permissions of both user/group principals and channels/portlets. Such features would likely make it easier to use more complex authorization policies with uPortal, but this complexity could make defining simple policies more complicated and error-prone. GridSphere's authorization mechanism lies somewhere between Clarens and uPortal. uPortal and GridSphere also allow for custom-defined authorization mechanisms, a extremely useful avenue for extensibility that Clarens lacks. Many organizations have some existing authorization services in place that they may wish to leverage for their newly deployed Grid portals. A notable area of concern affecting all three portals is secure default access configurations. The software should really be distributed with access disabled until administrators explicitly configure the access control policy and mechanisms as desired for their organization. In general, default configurations tend to be far more open.

C. Auditing

As a Java-based Grid portal framework, GridSphere relies on the widely used log4j [15] package for its logging support. Log4j allows the level of logging output (debug, info, warn, error, etc.) to be configured independently for the Java packages and classes in a project. Also configurable are the formatting/layout of the log messages and where they should be written to (file, standard output, etc.). By default, these messages are standard out which means that they get re-directed into Tomcat's general output log file (catalina.out) along with any other messages generated by Tomcat itself or any other servlets hosted by Tomcat. Due to a lack of documentation about what kind of message logging support is built-in to the various Java packages that make up GridSphere, it's difficult to determine how to configure GridSphere to generate the desired audit trails, or even if it's *possible* to get GridSphere to log the desired events. Curiously absent is the use of syslog capabilities in logging with use of syslog facilities like *auth*.

uPortal, like GridSphere, makes use of log4j for its logging support. As a result, it shares many of the same logging features and shortcomings with GridSphere. By default, uPortal logs info, warn and error messages to a file called portal.log. While this is an improvement over throwing portal log messages in together with other Tomcat log messages, uPortal also lacks documentation about what events and information can be recorded and what can't. So while fine-grained control over logging policy might be theoretically possible, in practice it would entail a laborious inspection of the uPortal source code.

Clarens taps the logging support of the Apache web server for its log messages. Apache can record accesses to pages and mod_python modules (both failed and successful). Within a Clarens module or service, support is provided to write messages to the Apache error log with any other errors generated by the Web server. Logging configuration possibilities seem to be limited to enabling or disabling the output of debug related messages, although what is included in this category isn't really clear.

Overall, the logging support provided by the portal frameworks seems largely intended to log debugging and error messages *rather than for the purposes of generated useful audit trails*. This is especially true in the case of Clarens, which lacks a general purpose, configurable logging mechanism. In contrast, the log4j library used by the other two portals is designed to be very general purpose and could conceivably be configured to generate detailed audit trails in a consistent format. The obstacle to accomplishing this with existing implementations is that the Java classes that perform interesting actions or events must consistently report detailed information of these events to the logging system. There does not appear to be a systematic effort in either GridSphere or uPortal to log information about interesting events for the purposes of auditing. This is particularly evidenced by the lack of documentation about what events or information even could be logged for each class or package. Without the ability to generate audit trails with the desired content, it is basically impossible to systematically detect, much less correct security vulnerabilities.

D. Session Management

The Java Portlet Specification [1] defines a *PortletSession* object that portlets are expected to use to store their session data. Although some of the implementation details of this session object are left to the portlet container, the specification does define the important aspects of session features and usage. Portlets within a *portlet application* share the same session object for each client. (A portlet application is a web application consisting of a collection of portlets and their deployment configuration, along with possibly other servlets, web pages, etc.) The session data in a PortletSession object cannot be shared between portlet applications. The basic interface provided by the session object is for setting and retrieving arbitrary attributes as Java objects. This session management capabilities of portlets are built on top of the Java Servlet session features and in fact, attributes set for a PortletSession must be stored in the *HttpSession* object of the containing portlet/web application. Servlet containers (such as Tomcat) have several choices with regard to how sessions are tracked. The most popular approach is probably through a session-tracking cookie named JSESSIONID, whose contents (a session identifier of some sort) would be presented by the web browser to retrieve session state for a client. Other session-tracking possibilities include using the built-in session support of SSL (for HTTPS connections) and URL rewriting, in which the session identifier manifests itself in the URL used for subsequent portal requests.

As GridSphere and uPortal are both portlet-based, they both rely on the portlet session management features. Largely this means that the session implementation is provided by the servlet container, which would be Tomcat in both cases. By default, Tomcat uses cookies for session tracking, although this is configurable through the Tomcat configuration file. Tomcat typically stores session data in memory within the Java Virtual Machine (JVM), but can also be configured to store session data persistently (in the filesystem or in a database) so that it survives container restarts.

Clarens relies on the session IDs generated as part of the SSL protocol for session-tracking purposes (recall that Clarens uses the SSL protocol over HTTPS or HTTP-Basic for authenticating clients). Session state can then be stored and retrieved from a database (TDB) using the client and server session IDs. The server is responsible for making sure that this pair of session IDs is unique for each unique IP address. Because of persistence of the database, session state should survive server restarts without incident and is shared across all services/applications a server hosts.

Contrasting the session management features of the different portals, we see one that's very simple and home-grown (Clarens) and another that's more sophisticated and widely used (the portlet/servlet model used by GridSphere and uPortal). With any security-sensitive software, well-tested implementations are generally preferred over custom-built solutions that tend to have a greater potential for security vulnerabilities. This is of particular concern for the Clarens implementation. Clarens' limited library support for service writers to manipulate session data is another possible source of problems. In general though, session management is a critical area, and regardless of the level of library support, services that deal with session data must do so carefully to avoid introducing security vulnerabilities.

V. Recommendations

Designing, implementing, configuring and deploying secure Grid portals is a significant challenge and as a result of the relative newness of this area useful guidance on is often lacking. Prior to the analysis of Grid portals described in this paper, we established a set of general, minimum security guidelines for portals [6]:

1. The portal project must provide contact information a person in charge of the portal's security.
2. If the portal has a domain name, it must have appropriate mailboxes as per RFC 2142.
3. The system running the portal must have a risk and vulnerability assessment every two years.
4. System logging must be enabled on the portal host system and duplicated on a central log server.
5. The system log/audit trail must include for each portal application: application name, authentication success or failure, remote host address, remote user (if identified, RFC 931), authenticated user identity, authentication token type (password, X.509 certificate, kerberos, etc.).
6. Login and accounting data must be saved for 90 days, minimum.
7. Portal audit trails should include service requests initiated to back-end resources.

Upon completing our analysis of these three state-of-the-art Grid portal systems, we believe that the Grid portals should be improved to meet the basic guidelines enumerated above, as none of the three systems was particularly strong in its ability to generate audit trails. Furthermore, we now augment our initial list by focusing on the requirements of authentication, authorization and session management:

Require user identification and strong initial passwords. Particular for Grid portals that multiplex onto a single Grid account, one area that could be exploited would be to register with fake credentials (not fake *Grid* credentials) in the hope of gaining access to the portal and, possibly, to the backend resources. Policy and procedures should be in place for the registration of portal users which include verification of identity of the registrant (as per FISMA [8], discussed earlier in this paper) and when the user registers only allow selection of a reasonably strong password which could be implemented by the use of the libcrack tools.

Require strong on-the-wire authentication mechanisms whenever possible. Password-based authentication, though convenient, is a common source of security vulnerabilities, often through compromised passwords. Additionally, forms-based authentication, which is probably the most popular implementation mechanism for password-based systems, is notorious for introducing vulnerabilities in system designs. Stronger authentication mechanisms such as PKI-based X.509 certificates, Kerberos, etc. should be used where possible, preferably as the only authentication mechanism.

Require HTTPS for secure connections. To reduce the possibility of stolen session cookies and other sensitive data exchanges between the client and server, an encrypted connection should be used. As Grid portals are generally Web-based and the vast majority of web browsers support https, this is the easiest solution. This also allows for authentication of the server, giving clients confidence in their use of the portal and reducing the risk of cross-site scripting attacks. HTTP (as opposed to HTTPS) should no longer be accepted in the community for such information exchanges -- by assuming that *everything* should be transmitted over HTTPS, it makes it much easier for users and deployers to recognize misconfigured servers.

Use secure default configurations. Software that is not secure out of the box may *never* be secure, even when it's deployed and operational. To their credit, in the past few years, Microsoft has been promoting secure default configurations as one of the best approaches for securing software. Grid portals should have default configurations that restrict access, especially to administrative functionality. Default accounts and passwords are also problematic and introduce vulnerabilities that can be easily exploited. Force administrators to set the security configuration they want for their organization *before* deploying the portal. In general, insecure default configurations are a serious problem afflicting the state of the art in portal frameworks.

Prefer well-established security implementations to custom-built ones. This recommendation applies to a number of different areas: authentication, authorization and particularly session management. If a widely-used, well-tested security code exists for one of these areas, try to use it. Custom mechanisms are an easy way to introduce vulnerabilities. If custom mechanisms are required, take great care in the implementation and rely on code reviews and other well-established techniques to reduce security related errors.

Design administrative interfaces that are easy to use correctly. Even if an authorization or authentication module is implemented securely, configuration mistakes can easily introduce vulnerabilities. Configuration should be kept simple and straightforward wherever possible to reduce this possibility.

Clearly document configuration procedures and parameters. The importance of secure configuration cannot be overstated. The Grid portals we analyzed had gaps in documentation that could easily lead to configuration mistakes and security holes.

Log every access to Grid credentials stored by the portal. To facilitate access to backend resources, a Grid portal will often store delegated Grid credentials on behalf of the user. As compromise of these credentials is quite serious and could suggest a widespread security vulnerability, all accesses (store, read, remove, etc.) should be securely recorded. The use of syslog auth facility for recording this would be a good choice.

Regularly audit security logs. Although how often it is appropriate to do so may depend on the organization, waiting until you a problem shows up elsewhere is usually too late. Once the systems to record appropriate security events are in place and configured (as suggested by the SDSC guidelines), the audit trails that are generated must be analyzed regularly to detect and fix problems.

Some of the above recommendations are more relevant for portal implementers, some for portal administrators and some for portal application developers. This should not be surprising, as it requires a concerted effort from each of these groups to produce a secure Grid portal.

VI. Conclusions and Future Directions

While the primary goal of Grid portals is clearly to make the Grid easier to use, the role of security in Grid portals cannot be overstated. A compromised Grid portal can have serious consequences for the backend Grid resources the portal makes available, so it is essential that Grid portal developers and deployers eliminate or reduce security vulnerabilities. In our analysis of the state of the art in Grid portal environments (GridSphere, OGCE/uPortal and Clarens), we found that they vary in the sophistication and complexity of their security features in the key areas we examined: authentication, authorization, auditing and session management. This only underscores the importance of configuration. Any of the portals we looked at could be configured relatively securely but is perhaps more likely to be configured without comprehensive protection. Careful examination of the documentation and a thorough understanding of the various components in a portal's architecture are critical to ensure proper deployment. Following the recommendations we establish from our analysis further safeguards Grid users and resources.

We hope to use the results of our analysis to motivate improvements in portal security. Of particular interest is the area of logging and auditing, as a system in which portal logging policy is easily specified and configured and logs are written to a secure centralized location would certainly be welcome. Likewise a system for automatically analyzing Grid-portal-specific audit trails for detection of suspicious behavior on a regular basis would be a significant step forward. Finally, we plan to complement this architectural-level study of Grid portals with an analysis of the security record of a real, deployed and working Grid portal such as a TeraGrid Science Gateway. Issued considered will include: actual user registration, validation, and delivery of user identifiers in practice; implementation details of mapping portal users to resource provider users; how data access is limited between portal users when single portal role accounts are in use on the backend resource providers; breaches, etc. This could further clarify the magnitude and nature of the risk Grid portals actually face.

REFERENCES

[1] Abdelnur, A. and Hepper, S., eds. “Java Portlet Specification.” JSR-168, Version 1.0, 7 October 2003. Available at http://jcp.org/aboutJava/ communityprocess/final/jsr168/index.html.

[2] Ali, A., et al. “JClarens: A Java Based Interactive Physics Analysis Environment for Data Intensive Applications.” Proceedings of ICWS, the International Conference of Web Services, San Diego, USA, 2004.

[3] Amin, K., Hategan, M., von Laszewski, G. and Zaluzec, N.J. “Abstracting the Grid.” Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004), pp. 250-257, La Coruna, Spain, 11-13 February 2004.

[4] “Apache Pluto.” Apache Software Foundation. Available at http://portals.apache.org/pluto. Accessed 24 March 2006.

[5] “Apache Tomcat.” The Apache Software Foundation. Available at http://tomcat.apache.org. Accessed 24 March 2006.

[6] “CIP/SDSC Portal Minimum Security Standards.” Draft v 1.1. Available at http://security.sdsc.edu/policy/PortalPolicy.html. Accessed 24 March 2006.

[7] Coward, D. and Yoshida, Y., eds. “Java Servlet Specification.” JSR-158, Version 2.4, 24 November 2003. Available at http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html.

[8] Federal Information Security Management Act (FISMA). NIST Special Publication 800-53

[9] “Hibernate: Relational Persistence for Java and .NET.” JBoss, Incorporated. Available at http://www.hibernate.org. Accessed 24 March 2006.

[10] “Information security.” From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Information_security. Accessed 24 March 2006.

[11] “Introducing JSON.” Available at http://www.json.org. Accessed 24 March 2006.

[12] “Java Authentication and Authorization Service (JAAS).” Sun Microsystems, Inc. Available at http://java.sun.com/products/jaas. Accessed 24 March 2006.

[13] Johnson, R., et al. “Spring – Java/J2EE Application Framework.” Reference Documentation, Version 1.2.7, Chapter 11: Data Access Using JDBC. Available at http://www.springframework.org/docs/ reference. Accessed 24 March 2006.

[14] “JSON-RPC Specifications.” Available at http://json-rpc.org/wiki/ specification. Accessed 24 March 2006.

[15] “Log4J Logging Services Project.” The Apache Software Foundation. Available at http://logging.apache.org/log4j/docs/index.html. Accessed 24 March 2006.

[16] “Mod_Python: Apache/Python Integration.” The Apache Software Foundation. Available at http://www.modpython.org. Accessed 24 March 2006.

[17] Novotny, J., Russell, M. and Wehrens, O. “GridSphere: A Portal Framework for Building Collaborations.” Proceedings of the 1st International Workshop on Middleware in Grid Computing, Rio de Janeiro, Brazil, 2003.

[18] Roberts, E., et al. “GPIR - Grid Portals Information Repository.” Available at http://www.tacc.utexas.edu/projects/gpir.php. Accessed 24 March 2006.

[19] Shirley, R., ed. “Internet Security Glossary.” IETF Newtork Working Group, RFC-2828. May 2000. Available at http://www.ietf.org/rfc/ rfc2828.txt.

[20] Steenberg, C. et al. “The Clarens Grid-Enabled Web Services Framework: Services and Implementation.” Proceedings of CHEP 2004, paper 184, 2004.

[21] “TDB: Trivial Database.” Available at http://sourceforge.net/projects/ tdb. Accessed 24 March 2006.

[22] “The Ten Most Critical Web Application Security Vulnerabilities.” The Open Web Application Security Project (OWASP). 2004 Update, 27 January 2004. Available at http://www.owasp.org/documentation/ topten.html.

[23] Tuecke, S., Welch, V., Engert, D., Pearlman, L., Thompson, M. Internet “X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile.” IETF Network Working Group, Standards Track, RFC 3820, June 2004. Available at http://www.ietf.org/rfc/rfc3820.txt.

[24] “uPortal: Evolving Portal Implementations from Participating Universities and Partners.” http://www.uportal.org. Accessed 24 March 2006.

[25] von Laszewski, G., Foster, I., Gawor, J. and Lane, P. “A Java Commodity Grid Kit,” Concurrency and Computation: Practice and Experience, vol. 13, no. 8-9, pp. 643-662, 2001. Software available at http://www.cogkit.org.

[26] “Web application.” From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Web_application. Accessed 24 March 2006.

[27] “Web portal.” From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Web_portal. Accessed 24 March 2006.