GitBAC: Flexible Access Control for Non-Modular Concerns

Mark Robinson, Jianwei Niu, and Macneil Shonle
University of Texas at San Antonio
One UTSA Circle
San Antonio, Texas 78249-1644
{mrobinso,niu,mshonle}@cs.utsa.edu

***Abstract*—Today's techniques for controlling access to software artifacts are limited to restricting access to whole files and directories. But when a company's access control policy does not match a project's existing physical modularization, these techniques require either an all-or-nothing approach or re-modularization of the files and directories. The increased maintenance overhead this brings to project administration can lead to unimplemented or insufficient developer access control and an increased risk of insider security incidents (e.g., theft of intellectual property). We have created a tool (GitBAC) to provide access control of software artifacts using a crosscutting concern instead of artifact modularization. Our method provides fine-grained access control of artifacts and accommodates flexible access control policies.**

***Index Terms*—Access control; crosscutting concerns; development tools.**

I. Introduction

In 2010, Sergey Aleynikov, working as a programmer for Goldman Sachs, transferred thousands of Goldman Sachs' proprietary software files with the intent to deliver the code to one of their competitors [1]. While Aleynikov undoubtedly required access to some part of the codebase to perform his programming duties, a tighter access control strategy for the software artifacts (e.g., source code) could have reduced the impact of this incident. A 2008 insider threat study estimates that 27% of the intrusion incidents reported during the summer of 2006 originated from insiders and 15% were of unknown origin [2]. The study recommends that organizations should carefully tune fine-grained access control so that users only have access to the data necessary to accomplish their duties.

Current methods to control access to software development artifacts include project separation and fake integration points (e.g., test-only libraries). These techniques involve physically separating a project's source code into modules, where each module belongs to a specific level of access. Both of these conventional approaches suffer from two problems. Firstly, modularization of software artifacts for the sake of access control further decomposes the project's modularization strategy. Thus, changes that were previously contained to a design-based module can become crosscutting changes, affecting multiple modules and requiring additional management to maintain consistency. The second drawback is the inflexibility of access control policy implementation, as the mechanisms for implementation are a component of the artifact storage environment's static access control model.

We have created a tool called *Git-Based Access Control (GitBAC)*, which implements a software artifact access control policy as a crosscutting concern. GitBAC is layered on top of the Git version control system in order to seamlessly provide support for access control for non-modular concerns. Developers use Git as they have before, either through the command line or integrated development environment, and are automatically granted read or write access only to the program fragments to which they have been granted access. GitBAC provides the flexibility to specify access granularity of an artifact fragment as coarse as the entire artifact or as fine as a single token. Artifact fragments promote greater code sharing between access-controlled groups and eliminate the additional maintenance overhead imposed by modularizing a codebase for access control. As our technique is not a physical modularization of the code, it can be easily modified to accommodate new access control models and policies without altering an existing modularization strategy. Lastly, implementing access control as a crosscutting concern may bring new parallel development opportunities, such as automatic mock generation for restricted fragments.

II. The GitBAC Approach

GitBAC, or Git-Based Access Control, is our concern-based implementation of software artifact access control using the Git version control system as an artifact store. GitBAC acts as a proxy between the Git client software and the Git server software and artifact repository. Git supports three communication protocols: HTTP, SSH, and the Git protocol (for read only). GitBAC only uses the SSH protocol, which is satisfactory for maintaining confidentiality within an access controlled software development project. While GitBAC is a standalone proxy, all of our implementation and testing occurs within Eclipse using the eGit plugin as our Git client. The Git client authenticates only to GitBAC, oblivious of the location and credentials necessary to access the Git repository. Thus, GitBAC is the client's only avenue of access to the software development artifacts. GitBAC is written in Java using the jGit and Apache MINA SSH libraries.

A. Overview of the GitBAC System

Figure 1 shows an overview of the eight primary components of GitBAC: the Git client interface, the access control

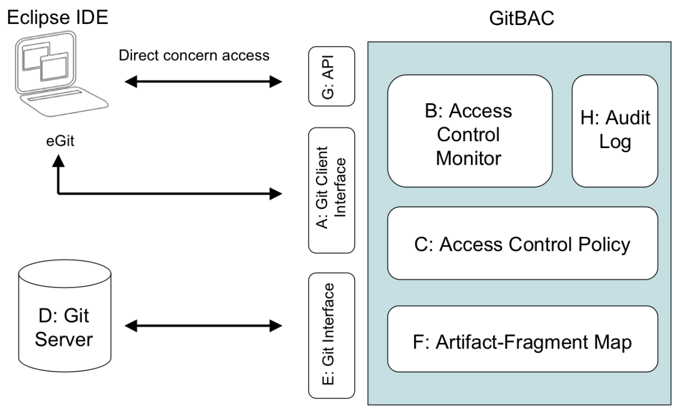

Fig. 1. GitBAC overview.

monitor, the implemented access control policy, the artifact store, the artifact access interface, the artifact-fragment map, the API, and the audit log.

a) The Git Client Interface: Git clients (e.g., eGit, the command line) access Git repository artifacts transparently through GitBAC's client interface using the SSH protocol (Figure 1(A)). GitBAC returns a data stream containing only the fragments of the artifacts that satisfy the access privileges. Access to fragment data occurs in different ways (e.g., importing a project, pushing a commit). Directory and package listings show only the artifacts to which the requesting user has access. Similarly, editor access requests may only read and write content to an artifact fragment where the user has appropriate access.

b) The Access Control Monitor: The access control monitor (Figure 1(B)) determines access privileges for the requested fragment based on the user's credentials and filters the data stream between client and artifact store based on the user's access privileges for a particular fragment.

c) Access Control Policies: GitBAC provides modular support for access control policies, allowing a variety of customizable policy implementations (Figure 1(C)).

d) Artifact Stores: The Git server's artifact store (Figure 1(D)) is maintained outside of GitBAC. However all client access to the artifact store occurs through GitBAC. Direct access between GitBAC and the artifact store is protected using secure, authenticated channels.

e) Artifact Access Interface: GitBAC communicates with the artifact store (Git) directly using SSH (Figure 1(E)). Only GitBAC has full access to all of the repositories maintained by the Git server.

f) Artifact-Fragment Map: Artifact-fragment transformations are created when fragments are modified according to the access control policy and served to the client (Figure 1(F)). When the client pushes updates through GitBAC to the Git server, the transformation is reversed so that the update may be applied to the original artifact.

g) The GitBAC API: GitBAC's Application Programming Interface (Figure 1(G)) provides a programmatic interface to the implemented access control concern. Using the API, one may customize importers, views, editors, and explorers to directly interact with the concern and its related artifact fragments.

h) Audit Log: For the purposes of forensics and testing, the access control monitor maintains an audit log of all client access attempts to artifacts, fragments, and the implemented access control policy (Figure 1(H)).

Group	Admin?	Developer?	Working Local?
Public	No	No	Any
DevRemote	No	Yes	No
DevLocal	No	Yes	Yes
Admin	Yes	Yes	Yes

TABLE I
GITBAC ACCESS CONTROL GROUP ATTRIBUTES.

B. Example GitBAC Access Control Policy

Git has no built-in access control mechanism. It relies on external tools or mechanisms inherent to the physical storage system (e.g., file system permissions) to implement an access control policy. Three tools provide repository or branch-level read/write access control: gitosis, gitolite, and gerrit. GitBAC differs from these tools as it applies an access control policy to software artifacts (the original artifacts and their changes) within the Git repository, crosscutting all development branches without modularizing the codebase.

We demonstrate a hierarchical Attribute-Based Group Access Control (ABGAC) policy in GitBAC. ABGAC allows finer-grained specification of jobs, tasks, and the degree of threat presented by each included entity [3]. Table I is a matrix of the attributes representing the GitBAC access control groups: Public, DevRemote, DevLocal, and Admin. The attributes *Admin?* and *Developer?* are specified for each authorized user in GitBAC's Access Control List (ACL) using the form: {*user name, Admin?, Developer?*}. Below is an example showing two GitBAC users: Bob with developer access and Sue with public access.

```
{Bob, false, true}
{Sue, false, false}
```

The attribute *Working Local?* is determined dynamically by GitBAC for each client request. If the client's IP address is within a certain range, *Working Local?* is *true*, otherwise *Working Local?* is *false*. If our example developer, Bob, communicates with GitBAC using an approved IP address, then Bob is a member of the DevLocal group, otherwise Bob is a member of the DevRemote group. Through the use of an ABGAC model, GitBAC can implement more expressive and flexible access control policies than with ACLs alone.

C. GitBAC Fragment Specification

GitBAC uses encoded comments within source artifacts to delimit fragments. These comments are protected from modification during the editing of source code artifacts. GitBAC ignores any fragment boundary that has been altered from the original when an update is pushed to it. Fragment boundaries may only be created and modified through API calls and take the form:

```
/*START @<fragment label>*/
fragment body
/*END @<fragment label>*/
```

We store the access rules for fragments in our ACL using the form:

```
<frag. label> [EX] read:{<group list>} write:{<group list>}
```

"read" indicates the read access operation and "write" indicates the write access operation. Absence from the braces implies denial of the particular operation unless the requestor inherits access through the access control group hierarchy (e.g., a "Public" group grants access to the particular operation to all requestors). "EX" is an optional argument that indicates an exclusive fragment. GitBAC's exclusive fragments protect target code from reverse engineering while allowing access-restricted developers to build and test. Exclusive fragments ignore inherited access privileges. Returning to our example developer Bob, we can specify a fragment, *foo*, with read access for both the DevLocal and DevRemote groups and write access for only DevLocal using the following rule in our ACL:

```
foo read: {DevLocal, DevRemote} write: {DevLocal}
```

D. GitBAC Interaction Types

GitBAC supports two types of access-controlled client interactions: downloading (importing) a project from GitBAC and pushing project updates back to GitBAC using the Team Remote Push operation. Importing a project uses the *git upload-pack* command and leaves the client with a crosscut of the complete project that is specific to the downloading client's access rights. Pushing updates back to the GitBAC server uses the *git-push* command.

a) Importing an Access-controlled Project using GitBAC: The Git client imports a project into Eclipse by cloning the project and its change history from the Git server to the local Eclipse workspace using the Git command *git upload-pack*. *git upload-pack* presents the Git client with a list of version branches from which the client may select and initiate the project download to the workspace. GitBAC inserts itself between the Git client and the Git server, so that Git projects are first crosscut by the downloader's access credentials. In this way, the downloader never sees any part of any artifact for which his/her access credentials do not allow. The sequence diagram in figure 2 illustrates the import dialogue between the Git client, GitBAC, and the Git server.

b) Pushing Updates to a Project through GitBAC: Pushing updates to GitBAC using *git-push* operates essentially in reverse of importing. The Git client uses *git-push* to send previously unsent updates to GitBAC. GitBAC examines each update from the client, expands the offset of the update in the artifact back to its original value, and determines if the update

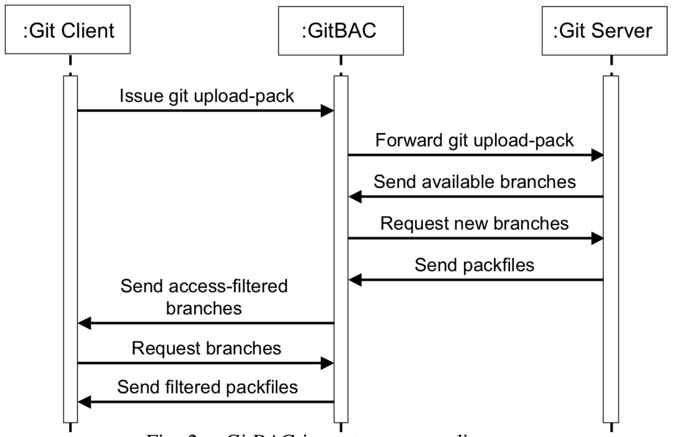

Fig. 2. GitBAC import sequence diagram.

pertains to a write-restricted fragment for the updating user. GitBAC discards any update that violates the access control policy and passes allowed updates to the Git server for normal application to the remote repository.

III. Related Work

Joshi et al. discuss the significance of the insider threat to software and the importance of access control as a protection mechanism [4]. They survey seven access control models, comparing them with respect to web-based applications and workflow management systems. They conclude that RBAC is the best candidate for suiting most enterprise access control web application requirements. However, they stress that RBAC needs to be extended in order to fully satisfy security policies. Pramanik et al. feel that a least privilege-based access control mechanism alone will not prevent insider attacks [5]. They implement a document access control model that enhances least privilege, considering the context in which the access request is made (e.g., currently opened documents and programs, prior requests). Our implemented access control policy allows the flow of higher access information to lower access fragments but our design can accommodate this types of policy enhancement.

Aspect-oriented techniques have been used to implement access control within software applications. Ramachandran et al. demonstrate the benefit of AspectJ to help locate multi-layer security points in Java programs [6]. Mens et al. use intensional program views to document and query high-level structural information in program development [7]. While these techniques do not specifically address access control of software artifacts during the development process, they can assist us in identifying and specifying crosscuts for access control implementation.

Lopez and van der Hoek survey and categorize many intensional and extensional concern-oriented approaches [8]. They suggest that future techniques incorporate more than one of the categories of approaches and support artifact types other than source code. An Eclipse navigation-based concern management tool, known as Mylyn extends the Eclipse views and editors to visualize the concern and supports several existing issue tracking databases, allowing support for multiple concurrent concerns [9]. Robillard and Murphy implement FEAT, a tool that uses structural dependencies in code and queries for locating and gathering concern-related fragments [10]. Harrison et al. describe the Concern Manipulation Environment (CME) as a tool that can construct and maintain arbitrary concerns via forward-engineering enumeration (extensional) and reverse-engineering queries (intensional) [11]. They promote artifact neutrality and representation of concern relationships and constraints. These techniques can potentially be used by our approach for access control concern construction and manual maintenance in GitBAC. Majid and Robillard describe NaCIN, an Eclipse tool that monitors and associates a developer's navigation/event activity and code structural dependencies with high-level concerns (i.e., developer-assigned tasks) [12]. They use Java methods as the basis of a fragment and store the concern model as XML. Robillard and Weigand-Warr present ConcernMapper, an Eclipse tool that assembles concerns based on a intensional technique using Eclipse code searches [13]. They support Java methods and member variables as the fragment elements. Nistor and van der Hoek introduce a tool for dynamic concern-driven programming [14]. Intensionally specified artifact fragments for development tasks (i.e., concerns) are dynamically updated as developers work with them using semi-automated heuristics and manual specification/correction. Our access control technique can transparently work in conjunction with these techniques so that the fragments they use are an access-permitted subset of the entire project. Thus, the set of fragments that a user possesses is an intersection of multiple concerns (access control being one of them).

Other techniques utilize concerns through mining code, dependency graphs, or change set information from a source code version control system. Ratanotayanon et al. describe a method for updating concern fragments using the diff utility: a conflict resolution tool native to version control systems [15]. Adams et al. implement and evaluate a history-based concern mining approach called COMMIT [16]. Their results indicate the usefulness in mining program elements at fine levels of granularity and the granularity depends on the type artifact being mined (C versus Java versus design documentation, etc.). Kozaczynski et al. describe an approach to automate transformations in source code using syntactic, semantic, and abstract concept-based knowledge via recognition rules and concept pattern descriptions [17]. These techniques may also be used in conjunction with our access control model to provide a crosscut of a codebase that suits the requestor's access level. Nita and Notkin propose a method for developers to specify code-level mappings between code alternatives instead of requiring redundant updating [18]. GitBAC protects target code through the specification of mutually exclusive fragments based on requestor credentials.

IV. Future Work

We see future research opportunities and contributions for GitBAC in the following areas:

a) Concern Assignment: We want to leverage existing techniques of concern creation to assist in the location and specification of fragments for access control concerns, e.g., Mylyn [9]. Even with the help of such tools, we feel that manual concern assignment for access control will continue to play at least a small role in accurate concern specification.

b) Mock Generation: GitBAC's mocks and stubs are manually specified in separate, exclusive fragments, but other opportunities for mock generation exist. We see the possibility of dynamically generating mocks in GitBAC for areas of restricted code flagged as "required for testing".

c) Security Leak Detection: Even with an access control policy in place, software development efforts may still suffer accidental and intentional fragment access violations. We can provide an extensible mechanism for detection and notification of violations of an access control policy and GitBAC's audit history can play an important role in determining the nature, magnitude, and accountability of a security incident.

References

[1] J. Lynch, "Programmer indicted in theft of goldman software," *The New York Times*, pp. 38–47, Februrary 2010.

[2] E. Kowalski, D. Cappelli, and A. Moore, "Insider threat study: Illicit cyber activity in the information technology and telecommunications sector," Carnegie-Mellon University Pittsburgh PA Software Engineering Inst, Tech. Rep., 2008.

[3] M. Bishop, S. Engle, S. Peisert, S. Whalen, and C. Gates, "We have met the enemy and he is us," in *Proceedings of the 2008 workshop on New security paradigms*, 2008, pp. 1–12.

[4] J. B. D. Joshi, W. G. Aref, A. Ghafoor, and E. H. Spafford, "Security models for web-based applications," *Commun. ACM*, vol. 44, pp. 38–44, 2001.

[5] S. Pramanik, V. Sankaranarayanan, and S. Upadhyaya, "Security policies to mitigate insider threat in the document control domain," in *ACSAC*, 2004, pp. 304–313.

[6] R. Ramachandran, D. J. Pearce, and I. Welch, "AspectJ for multilevel security," in *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2006, pp. 13–17.

[7] A. Kellens, F. Pluquet, and R. Wuyts, "Co-evolving code and design with intensional views - a case study," in *Computer Languages, Systems and Structures*, 2006, pp. 140–156.

[8] N. Lopez and A. van der Hoek, "An agenda for concern-oriented software engineering," in *FoSER*, 2010, pp. 217–221.

[9] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *AOSD*, 2005, pp. 159–168.

[10] M. P. Robillard and G. C. Murphy, "Feat: a tool for locating, describing, and analyzing concerns in source code," in *ICSE*, 2003, pp. 822–823.

[11] W. Harrison, H. Ossher, S. Sutton, and P. Tarr, "Concern modeling in the concern manipulation environment," in *MACS*, 2005, pp. 1–5.

[12] I. Majid and M. P. Robillard, "Nacin: an eclipse plug-in for program navigation-based concern inference," in *Proceedings of the OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 70–74.

[13] M. P. Robillard and F. Weigand-Warr, "Concernmapper: simple view-based separation of scattered concerns," in *Proceedings of the OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 65–69.

[14] E. C. Nistor and A. v. d. Hoek, "Explicit concern-driven development with archevol," in *ASE*, 2009, pp. 185–196.

[15] S. Ratanotayanon, S. E. Sim, and D. J. Raycraft, "Cross-artifact traceability using lightweight links," in *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, 2009, pp. 57–64.

[16] B. Adams, Z. M. Jiang, and A. E. Hassan, "Identifying crosscutting concerns using historical code changes," in *ICSE*, 2010, pp. 305–314.

[17] W. Kozaczynski, J. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Trans. Softw. Eng.*, vol. 18, pp. 1065–1075, 1992.

[18] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *ICSE*, 2010, pp. 205–214.