How to securely break into RBAC: the BTG-RBAC model

Ana Ferreira, David Chadwick
School of Computing
University of Kent
Canterbury, UK
{af84,d.w.chadwick}@kent.ac.uk

Pedro Farinha, Ricardo Correia
CINTESIS
Faculty of Medicine
Porto, Portugal
{pedro_fa, rcorreia}@med.up.pt

Gansen Zao
School of Computer Science
South China Normal University
China
zhaogansen@gmail.com

Rui Chilro
Service of Informatics
Faculty of Nutrition and Food Sciences
Porto, Portugal
rchilro@fcna.up.pt

Luis Antunes
Institute of Telecommunications
Faculty of Science
Porto, Portugal
lfa@ncc.up.pt

Abstract—**Access control models describe frameworks that dictate how subjects (e.g. users) access resources. In the Role-Based Access Control (RBAC) model access to resources is based on the role the user holds within the organization. RBAC is a rigid model where access control decisions have only two output options: Grant or Deny. Break The Glass (BTG) policies on the other hand are flexible and allow users to break or override the access controls in a controlled and justifiable manner. The main objective of this paper is to integrate BTG within the NIST/ANSI RBAC model in a transparent and secure way so that it can be adopted generically in any domain where unanticipated or emergency situations may occur. The new proposed model, called BTG-RBAC, provides a third decision option BTG, which grants authorized users permission to break the glass rather than be denied access. This can easily be implemented in any application without major changes to either the application code or the RBAC authorization infrastructure, apart from the decision engine. Finally, in order to validate the model, we discuss how the BTG-RBAC model is being introduced within a Portuguese healthcare institution where the legislation requires that genetic information must be accessed by a restricted group of healthcare professionals. These professionals, advised by the ethical committee, have required and asked for the implementation of the BTG concept in order to comply with the said legislation.**

Keywords-Access control model; NIST Core RBAC; Break The Glass; Obligations

I. INTRODUCTION

Access control models describe frameworks that dictate how subjects (e.g. users) access resources. In the Role-Based Access Control (RBAC) model a set of controls is defined in order to determine how subjects and resources interact. The RBAC model allows access to resources based on the roles the user holds within the organization [1]. This model has been widely used and accepted to enforce access control in many domains and so an American standard has been created in order to formally define a fundamental and stable set of RBAC features and components [2]. Although flexible and easier to manage within large-scale organisations than discretionary access control lists, RBAC is usually a rigid model where access control decisions have only two output options: Grant or Deny.

There are some cases when this is not enough. For traditional access control models there is usually the assumption that access permissions are known in advance, and that the rules have been set up correctly, but in real settings, errors are made and unanticipated or emergency situations may occur. This mandates that a more flexible and adaptable approach be adopted [3]. In such cases as these, a *Break The Glass* (BTG) policy can be used in order to *break* or override the access controls in a controlled manner (the name is BTG because it is a similar process to breaking the glass on a fire door or a fire alarm). The concept is not new, it has been studied and introduced in several domains [3-6]. A BTG policy should allow a user to override the rules stated by the access control manager and access what he requests, even though he was not previously authorized to do it. But in so doing, other BTG rules come into play (such as obligations to undertake predefined actions and enforcement of decisions [7]) which may monitor, record or report the user's actions, thus making him responsible and oblige him to justify what he did. We propose to support break the glass policies by introducing a third option, BTG, to supplement the existing Grant and Deny responses in RBAC. BTG will be returned by the policy engine when the user is not currently authorized to access the resource (so Grant is not appropriate), but neither is he absolutely forbidden access to it (so Deny is not appropriate either). Instead, the BTG policy says that this class of users is entitled to break the glass if they are prepared to face the consequences for this.

The main objective of this paper is to integrate BTG within the NIST RBAC model in a transparent and secure way so that it can be adopted generically in any domain where unanticipated situations may occur. We call this the BTG-RBAC model.

This paper is organized as follows. Section II describes, in more detail, the existing concepts of BTG, obligations, the NIST/ANSI RBAC core model, as well as the RBAC core model augmented with obligations. Section III describes our proposed enhancement of the obligation augmented RBAC model to include BTG (the BTG-RBAC model). Section IV discusses the validation of the proposed model as well as its future implementation and evaluation in a real medical environment. Section V concludes the paper.

II. Background Information

A. *Break The Glass (BTG)*

Traditional access control policies are designed to be restrictive. The assumption is that users prefer to have unrestricted access to everything and so need to be controlled. Consequently, access control implementations focus mainly on avoiding security breaches and consequently they do not always best serve the user's needs and purposes. Access control policies that are instead defined with maximum freedom of access and, at the same time, maximum user responsibility for any exceptional actions taken, are preferable to traditional ones. By *maximum freedom* we mean the system must provide mechanisms for the users to access the requested information at all times, whenever it is needed. By *maximum user responsibility* we mean the system must provide mechanisms to show the user (who takes an exceptional action) an alert message making him aware that he is trying to access information he is not authorized to see. This makes him responsible for what he is doing and all the actions he may subsequently take; the system must provide mechanisms to automatically notify all responsible parties so that the user's actions can be justified afterwards to them [5].

As an example, an application domain where BTG is an essential feature is healthcare.

According to legislation, the HIPPA act specifies the need for BTG [8] as is described in [6]. BTG is needed when normal access controls to processes are insufficient and an emergency access control mechanism is required. Examples of emergency situations that might require BTG could be account problems (e.g. a user has not been given the proper roles or permissions) or authorization problems (e.g. an emergency situation such as hurricane Katrina thrusts an individual into a role that lacks sufficient access rights to perform the needed actions). A similar concept is the one described in the NHS documentation as *break the seal* on sealed documents [9]. The idea is that patients have the right to seal information. They can place access restrictions on parts of their medical records. An email alert is raised when the seal is broken and a privacy officer investigates if the action taken was justifiable or not. Moving from legislation to practice, [5] presents a good example where BTG is needed. It describes an access control policy that was defined by healthcare professionals (mainly doctors who stated that BTG was a very important feature to be integrated within the policy and the system that was to be implemented).

BTG is a required aspect both in terms of generic and theoretical as well as practical issues, so it needs to be integrated in a transparent and modular way in the domain where it is needed and within the access control policy and model that is developed within any information system.

B. *Related Work*

Research has been progressing in access control in order to integrate more flexibility and adaptability to access control policies. The Risk-Adaptable Access Control (RAdAC) model is an example that recognizes in some situations, the consequences to an organization of not sharing information might be worse than of sharing it [10]. The security risk has to be balanced against the operational need. The main difference from traditional models is that RAdAC provides flexibility to adapt access control decisions according to the situation at hand. Security policy grants or denies can be reversed according to the operational need at the time of the requested access.

Similar work has been done in the healthcare environment as this also requires more dynamic characteristics than access control policies usually allow. Most existing implementations solve this issue with exception handling mechanisms. But this may not be enough for healthcare applications which often have special requirements that need to be better studied [11]. Consequently, the same researchers decided to study the access control requirements in healthcare by analyzing user access logs from systems with extensive use of exception-based access control [12]. They found that the use of exception mechanisms was quite common but was not the correct way to perform access control in healthcare. They concluded that there was a need to reduce the usage of exception handling mechanisms. The work on BTG described in this paper is one solution to this problem.

Including BTG as a generic extension of access control models is presented in [13]. This work provides a means of specifying generic BTG policies using secureUML for an architecture that is based on java and XACML.

C. *Obligations*

Another important aspect closely related with BTG is obligations. Obligations are operations that are triggered and need to be compulsorily performed when an action is taken. They are duties, which are associated with privileges (or permissions). So when an operation is performed on an object, the obligations that are associated with that permission are activated and performed along with the operation. In the case of RBAC authorisation an obligation is performed when a response from the authorisation infrastructure is received and there are obligations associated with the request to be performed by the user.

Prior research has been undertaken which refers to the need for obligations either to provide for data integrity [14] or to require the performance of tasks associated with users' actions [15] or to coordinate authorization decisions in a distributed system [16]. Policies with obligations are formalized where obligations can be performed before or after the user is granted the requested permission. Obligations can be specified and managed within the policy [17] [18].

Obligations have been integrated into the CORE RBAC model [7] in a transparent and secure way. This augmented model is capable of providing obligations for both Grant and Deny responses and it is the appropriate model to use to integrate the BTG features.

D. The ANSI Core RBAC Model

1) Core RBAC Model

The ANSI Core RBAC model consists of five basic elements, which are the USERS, ROLES, OPS (operations), OBS (objects), and SESSIONS, and five relations, which are (Fig. 1):

- **UA**: User-Assignment $\subseteq$ USERS x ROLES, a many-to-many mapping user-to-role assignment relation
- **PA**: Permission-Assignment $\subseteq$ PRMS x ROLES, a many-to-many mapping permission-to-role assignment relation.
- **U-S**: user_sessions (u:USERS) $\rightarrow$ 2^{SESSIONS}, the mapping of user *u* onto a set of sessions
- **S-R**: session_roles (s:SESSIONS) $\rightarrow$ 2^{ROLES}, the mapping of session *s* onto a set of roles
- **PRMS**: $2^{(\text{OPS x OBS})}$, the set of permissions;

  Op:(p $\in$ PRMS) $\rightarrow$ {op $\subseteq$ OPS}, the permission-to-operation mapping, which gives the set of operations associated with permission *p*;

  Ob:(p $\in$ PRMS) $\rightarrow$ {ob $\subseteq$ OBS}, the permission-to-object mapping, which gives the set of objects associated with permission *p*.

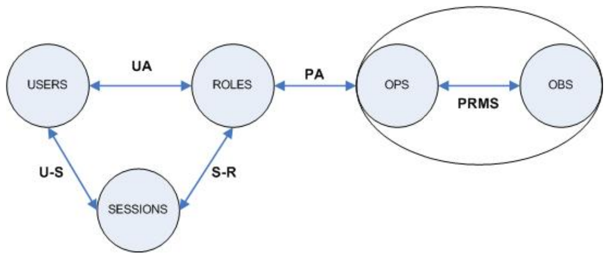

Figure 1 - The Core RBAC Model [2].

The authorization decision making function *CheckAccess* describes how a decision is made within the Core RBAC model by taking as inputs the current session, the requested operation and the target object and returns a Boolean value as a result to indicate whether the request is authorized or not.

$$\text{CheckAccess :SESSIONS}\times\text{OPS}\times\text{OBS}\longrightarrow\text{BOOL}$$

$$\text{CheckAccess(s,op,ob)}= \quad (\exists r \in ROLES: r \in S-R(s) \wedge ((op,ob),r) \in PA)$$

The *CheckAccess* function checks if a role *r* can be mapped for the current session *s*, such that *r* has been allocated the permission to perform the operations *op* on the objects *ob*. If such a value exists, the function returns TRUE (Grant) if not, FALSE (Deny) will be returned.

The steps to access a resource by a user with the Core RBAC model are (Fig. 2):

1. The user sends an *access application resource* request to the application
2. The application contacts the Authn Service to authenticate the user
3. The Authn Service returns the authenticated identity of the user to the application (If authentication fails, a reject message is sent from the application to the user and the request terminates here)
4. The application calls the RBAC policy engine passing the session details, the requested operation and requested object (CheckAccess)
5. The RBAC engine returns Grant to the application (or Deny, in which case a reject message is sent from the application to the user and the request terminates here)
6. The application makes the requested operation to the resource
7. The resource returns the results to the application
8. The application returns the results to the user.

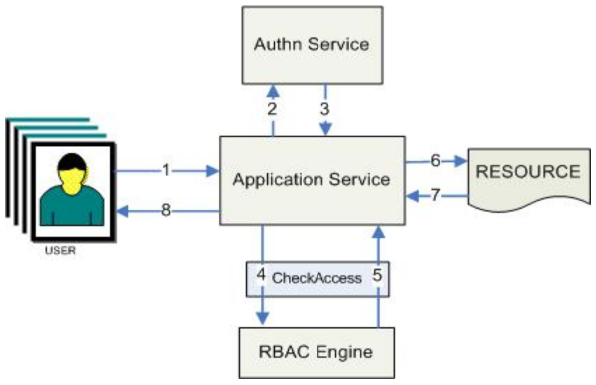

Figure 2 - Core RBAC interactions diagram.

2) Core RBAC with Obligations

In order to augment the Core RBAC model with obligations a new basic element OBLGS is introduced in [7], which is the set of valid obligations. The PRMS relation is replaced by a new relation OPRMS defined as $OPRMS = PRMS \times 2^{OBLGS}$. The PA relation is also replaced by a new relation, the permission-obligation assignment relation (POA) which is defined as follows:

$$POA \subseteq OPRMS \times ROLES$$

oprm ∈ OPRMS, and *oprm* is an obligation augmented permission: *oprm = (r,prm,oblgs).* This specifies if the permission *prm* is granted to role *r* through *oprm* and is exercised by the role *r*; the set of obligations *oblgs* must be fulfilled (Fig. 3). [7] also describes how the RBAC model can be augmented with obligations on deny, but this is not explained here due to space limitations.

In order to retrieve the obligations along with the authorization decisions, the *CheckAccess* function must be enhanced to:

$$\text{CheckAccess:SESSIONS}\times\text{OPS}\times\text{OBS}\longrightarrow\text{BOOL}\times 2^{\text{OBLGS}}$$

The possible results from *CheckAccess* are now:

- (FALSE, Ø) |→ DENY access to resource
- (FALSE, 2^{OBLGS}) |→ DENY access to resource AND perform Obligations on Deny
- (TRUE, Ø) |→ GRANT access to resource
- (TRUE, 2^{OBLGS}) |→ GRANT access to resource AND perform Obligations on Grant

The steps to access a resource by a user with the Core RBAC augmented with obligations are the same as described in Core RBAC with the added step of retrieving and performing obligations, if they exist.

III. THE BTG-RBAC MODEL

The BTG-RBAC model includes break the glass functionality within the RBAC engine (Core RBAC model with BTG) assuming we have a state based engine in order to alter the BTG state of a policy rule. With this assumption, the changes to include BTG are minimal and are described in the following sections.

A. The Simple BTG-RBAC Model

In order to integrate BTG within the Core RBAC model we introduce the BTG-RBAC engine, which holds the BTG state of each permission in the system. Initially the BTG state of each permission is set to FALSE, but it can be set to TRUE if there is a policy rule that allows a user to perform the break the glass operation O^{BTG} on a particular resource.

BTG-RBAC is accessed via an enhanced *CheckAccess* procedure, which we have called *CheckBTGAccess.* It returns one of three decision values to the application: Grant, Deny or $P^{BTG(r,op,ob)}$. $P^{BTG(r,op,ob)}$ grants the role *r* permission to break the glass for the operation *op* on the object *ob*. $O^{BTG(op)}$ is defined as the "break the glass" operation on a resource object for a defined operation *op*. The possible results for *CheckBTGAccess* are:

(GRANT)	**IF** *there is a rule granting the user's active role either the necessary permission, or permission if the BTG state is TRUE and the BTG state is actually TRUE*
($P^{BTG(r,op,ob)}$)	**IF** *there is a rule granting the user's active role permission if the BTG state is TRUE but the BTG state is FALSE*
(DENY)	***Otherwise***

An example of how a simple BTG policy might be specified by a security administrator is as follows (Table I):

TABLE I – EXAMPLE OF A SIMPLE BTG-RBAC POLICY.

Role	Operation	Object	BTG
r1	read	obs1	
r2	read	obs1	TRUE

The two policy rules described in Table I state that role *r1* is allowed to perform the *read* operation on the object *obs1*, and role *r2* is only allowed to perform the *read* operation on object *obs1* if the "glass is broken" i.e. the BTG state is TRUE. Implicit in this rule is the assumption that role *r2* is allowed to perform the "break the glass" operation $O^{BTG(read)}$ on object *obs1*. This implicit rule does not need to be stated explicitly in the simple policy model. The model is easy to understand and the rules are simple to write. When *checkBTGAccess(s,op,ob)* is called for (r2,read,obs1), then if the BTG state is TRUE, Grant will be returned, else $P^{BTG\ (r2,read,obs1)}$ will be returned. If the user decides to take responsibility to break the glass, then when *CheckBTGAccess*(s,op,ob) is called for (r2, $O^{BTG(read)}$,obs1) then GRANT will be returned (as per the implicit rule) and the BTG state variable for the permission assignment will be set to TRUE by the RBAC engine.

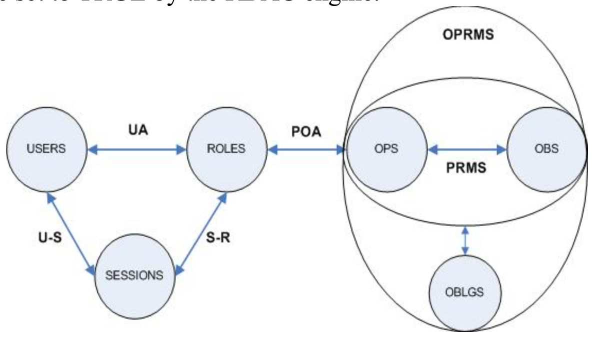

Figure 3 - The Core RBAC model with Obligations.

However, there are a number of limitations with the simple policy model. The first limitation is the implied rule and its corresponding assumption that there is one BTG state variable for every permission assignment i.e. Role/Operation/Object combination. This is somewhat inflexible in practice, since it would not allow one role to break the glass on a resource and thereby grant another role (or indeed all roles) access to the resource (as can happen when the glass is broken on a hotel fire door). Another limitation of the simple model is that the BTG-RBAC system does not know when or how to set the BTG state variable back to FALSE. A final limitation is that in most real life situations, when a subject does break the glass, one would normally want to place some obligations on this action, such as notify the manager, write to an audit trail and so on. The following section will address the limitations of the simple model.

B. *The Complete BTG-RBAC Model*

Addressing the limitations that were mentioned previously leads us to a more complex model where: new rules are added describing who is allowed to perform the $O^{BTG(op)}$ operation on a resource (this relaxes the enforced binding between the role that is allowed to break the glass and the role that is allowed to access the resource if the glass is broken); obligations are added to the $O^{BTG(op)}$ permission, allowing administrators to define arbitrary actions that must be performed when the glass is broken; the granularity of the BTG state variable can be varied from the fixed one state per permission assignment i.e. Role/Operation/Object combination; and rules can be added saying how the BTG state variable is reset to FALSE.

An example of the more sophisticated BTG-RBAC model is exhibited in the policy in Table II.

TABLE II – EXAMPLE OF A COMPLEX BTG-RBAC POLICY.

Role	Operation	Object	BTG	Obligations
r1	read	obs1		
r2	read	obs1	BTGi	
r2	$O^{BTG(read)}$	obs1		oblgs2_btg [Notify Manager; Write to Audit; Reset BTGi to FALSE after 30 mins]
r3	read	obs1	BTGi	oblgs3_btg [Write to Audit]
r4	$reset^{BTG}$	BTGi		

BTGi is a state variable of n dimensions over role, operation, object and environment i.e. BTG(r,op,ob,env) and will be described more fully in section III.D. Table II states that Role *r1* is allowed to *read obs1*, Role *r2* is allowed to *read obs1* if the break the glass variable *BTGi is TRUE*, Role *r3* is allowed to *read obs1* if the break the glass variable *BTGi is TRUE* but the system must perform one obligation simultaneously with granting access, Role *r2* is allowed to "break the glass" for *reading obs1* but the system must perform three obligations if *r2* does this, and Role *r4* is allowed to set the *BTGi state variable to FALSE*. The function *CheckBTGAccess* will now return the following results augmented with obligations:

CheckBTGAccess:
$$SESSIONS \times OPS \times OBS \longrightarrow \{T,F,P^{BTG}\} \times 2^{OBLGS}$$

***CheckBTGAccess*(s,op,ob) =**

(GRANT, 2^{OBLGS}) **IF** *there is a rule granting the user's active role either the necessary permission, or permission if the BTGi state is TRUE, and the BTGi state is actually TRUE*

(P^{BTG}) **IF** *there is a rule granting the user's active role permission to break the glass*

(DENY, 2^{OBLGS}) ***Otherwise***

C. *Formal Definition*

Defining now formally the new relations of BTG-RBAC from the Core RBAC model with obligations that was introduced in section II.D.2, we need to consider the set BTGS of BTG variables as defined in the previous subsection, the permission obligation assignment (POA) relation is modified to **POA_BTG** in the new BTG-RBAC model:

$$PRMS\_BTG = OPRMS \times 2^{BTG}$$

$$POA\_BTG = PRMS\_BTG \times ROLES$$

Again, the relation OPRMS is also used in the new model where:

OPRMS $\subseteq$ OPRMS_BTG AND
OPRMS_BTG $\subseteq$ PRMS x BTGS x 2^{OBLGS}

The relation POA_BTG for the policy in Table II would look like:

POA_BTG = {<r1, read, obs1, {}, {}> ; <r2, read, obs1, BTGi, {}> ; <r2, $O^{BTG(read)}$, obs1, {}, oblgs2_btg> ; <r3, read, obs1, BTGi, oblgs3_btg> ; <r4, $reset^{BTG}$, BTGi, {}, {}}.

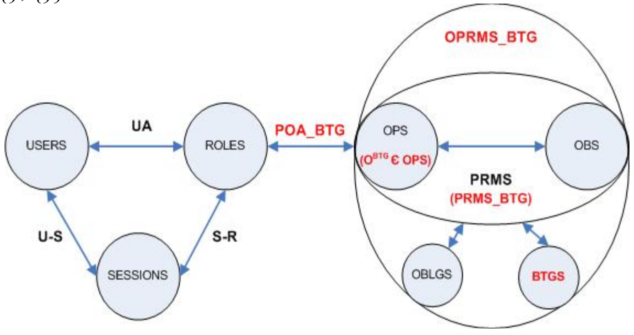

Figure 4 – The BTG-RBAC Model.

The new architecture of the BTG-RBAC model is presented in Fig. 4.

D. Handling the BTG State

Concurrently with a successful O^{BTG} operation there is the need to set the BTGi state variable to TRUE (if is not already set). The BTG-RBAC model is consequently state based as it needs to remember the state of the BTGi state variables. The writer of the BTG-RBAC policy determines the dimensions of the BTGi state variables. They could be based on the user's roles, the operation, the object, or environmental parameters such as a time period, etc. An example of various BTGi state variables is given in Table III.

TABLE III – EXAMPLE OF BTGI STATE VARIABLES.

Role	Operation	Object	Environment
r2	Read	obs1	30 minutes
*	*	obs2	Daily
*	Write	obs1	*

The first BTG state variable is dependent upon all 4 dimensions, thus it is only applicable for role *r2* performing operation *Read* on object *obs1*. Because it is time dependent, the BTG-RBAC engine will automatically create a new state variable every 30 minutes. If desired, the administrator could define a different BTG state variable for the same role (r2) performing a different operation (say Delete) on the same object in the same time periods. The second BTG state variable is for all operations by all roles on object *obs2* on a daily basis i.e. there is a different state variable for each day. If any role has permission to break the glass for any operation on *obs2*, it means that once this is done then the state *BTG(obs2)* will be set to TRUE so that any other role with any other break the glass permission on *obs2* will have had the glass broken for them. The third BTG state variable is for use by all roles with *Write* permission to object *obs1* for all environments. If a role breaks the glass for writing to obs1, this will not affect any role with permission to Read obs1. With the use of an n dimensional BTG state array, BTG can be defined in a fine-grained way so that a user can perform BTG with a combination of roles, operations, objects and environmental parameters.

Resetting the BTGi State Variables

BTG state variables require a service that can reset each BTGi state variable to FALSE. This can be done automatically, semi-automatically or manually. All three ways are needed. Automatic resetting means that the BTG-RBAC engine itself resets the BTGi state variable to FALSE after a specified event has occurred. The event must be specified by the administrator when creating the BTG-RBAC policy. Example events could be the expiration of a time period such as 30 minutes, or after a certain number of accesses have been made while the BTGi state was TRUE. Automatically resetting the BTGi state to FALSE controls the availability of a resource once the glass has been broken, and requires a second breaking of the glass after the specified event has occurred, before additional accesses can be granted.

We do not dictate how these events should be specified for the BTG-RBAC engine, or which events should be supported by a BTG-RBAC engine. We leave this to each BTG-RBAC engine supplier to specify for themselves.

Semi-automatic resetting of the BTGi state is similar to automatic resetting, but it is carried out in a standardised way by a system component that is external to the BTG-RBAC engine. For this we specify a new function *resetBTGstate (BTGi)* that must be supported by the BTG-RBAC engine. Any system component may call this function to reset the *BTGi* variable to FALSE. In our implementation we use an obligations service as the external system component. Using obligations, the security administrator sets an obligation in the policy rule that describes when the BTG state is to be reset. The events for when this occurs can be similar to the ones for automatic resetting. For example, obligations could be defined as follows: *Obligation set BTGi to FALSE after 30 minutes* or *Obligation set BTGi to FALSE after 3 BTG accesses*. These obligations are returned within 2^{OBLGS} once the user chooses and is allowed to perform O^{BTG}. The obligations will be performed when the events that are defined occur ("after 30 minutes" or "after 3 BTG accesses"). The middle row of the example policy in Table 2 gives an example of an obligation that will reset the BTGi state to FALSE 30 minutes after it is set to TRUE.

Manually resetting the state means that human intervention must occur before the BTGi state is set to FALSE, and policy rules should specify who is allowed to reset the state. This requires a new operation for resetting the state, which we have defined as the $reset^{BTG}$ operation ($reset^{BTG} \in OPS$). This operates on the BTGi state variable as the resource object. The last row of the example policy in Table II gives an example of a policy rule for manually resetting the BTGi state to FALSE. The BTGi state will only be reset after the permitted role, *r4*, issues the $reset^{BTG}$ operation on the BTGi object.

E. Steps to Perform BTG

The necessary steps for a user to perform BTG within a resource in the new BTG-RBAC model, assuming that the BTG state is initially FALSE, are as follows (Fig. 5):

1. The user tries to access a resource he/she is not authorized to
2. The authentication service validates the user's credentials
3. The authentication service returns the authenticated identity of the user

   (In the case where the authentication service fails, a reject message is sent from the application to the user and the request terminates here)

4. **If the user is authenticated, the application calls the BTG-RBAC policy engine passing the session details, the requested operation and requested object (*CheckBTGAccess*):**

   In the case where *there is a policy rule granting access to the object*, *CheckBTGAccess* returns Grant, so it goes to step 9;

   In the case where there is a policy rule granting O^{BTG} access to the object the BTG-RBAC engine returns P^{BTG} as the decision value;

   In all other cases CheckBTGAccess returns Deny and the request terminates here;

5. **The application can now ask the user if he/she wants to O^{BTG} on that resource. If the user chooses to O^{BTG} (giving a reason for it, if applicable) go to the next step.**

   (In the case where the user chooses not to O^{BTG} the original request terminates here)

6. **The application calls the BTG-RBAC policy engine passing the session details, the requested operation ($O^{BTG(op)}$) and the requested object (*CheckBTGAccess*):**

   The BTG-RBAC policy engine checks the policy, sees the operation is granted, sets the BTG_i state variable to TRUE and returns any obligations associated with the $O^{BTG(op)}$ operation (e.g. notify a responsible manager, write to an audit) to the application along with the GRANT response

7. **The application performs the returned obligations and the user is again shown the option to access the resource he requested and selects it.**

8. **The application calls the BTG-RBAC policy engine passing the session details, the original requested operation and object (*CheckBTGAccess*):** *CheckBTGAccess* returns Grant as the BTGi state variable is already set to TRUE

9. 10 & 11 The application makes the requested operation to the resource that returns the results to the application service, which gives them to the user.

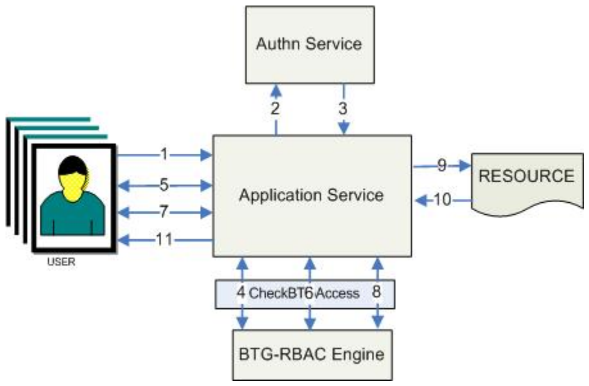

Figure 5 - The BTG-RBAC interaction diagram.

IV. IMPLEMENTATION OF BTG IN A REAL SETTING

This paper shows how BTG can be added to the Core RBAC model. The BTG-RBAC model can be easily implemented within a state based RBAC authorization infrastructure such as PERMIS [19], where the Core RBAC model with obligations has already been integrated and consolidated [7]. Since May 2009, the BTG concept has been implemented and is in use in a real medical setting in the second largest hospital in Portugal (Hospital S. João - HSJ) within a Virtual Electronic Patient Record (VEPR) [20]. This VEPR was implemented in 2004 and integrates an average of 3000 medical reports per day from 11 departments and is accessed on a daily basis by 1000 medical doctors.

The Portuguese legislation requires that genetic information must be accessed by a restricted defined group of healthcare professionals [21]. To comply with this legislation, the Ethical Committee from HSJ requested the implementation of BTG in the VEPR to restrict access to genetic information within the collected reports. This was implemented initially in a proprietary way, without using a RBAC engine and before we developed the BTG-RBAC model, because there was an urgency to enforce the legislation that came out in 2005, on a system that was being used since 2004. From this early implementation experience we realised that standardising BTG through the BTG-RBAC model and implementing it in an application independent way, via a BTG-RBAC engine, would make the work very much quicker for all subsequent applications. So our next step was to add BTG to PERMIS, an existing RBAC engine, and define the BTG-RBAC model that is presented here.

Below we present some preliminary results from our real environment where BTG-RBAC is implemented in a proprietary way (Fig. 6). When a user has "break the glass" permission they are asked the question "Do you want to break the glass?" to which they can answer "Yes" or "No". Table IV shows that with a few more than 3 months' use, BTG is a necessary tool to control who may access more sensitive information.

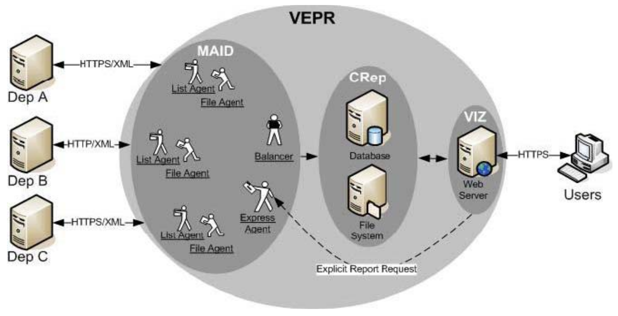

Figure 6 - General architecture of the EMR system showing the MAID, the VIZ modules and the CRep.

The core of the EMR system is composed of three modules (VIZ – Viewing modules, MAID - Multi-Agent system for Integration of Data, and CRep – Central repository) which are presented in Fig. 6. MAID collects clinical reports from various hospital departments (e.g. DIS A and DIS B), and stores them on a central repository (CRep) consisting of a database holding references to these reports. After searching the database, the users can access the integrated data of a particular patient through a web-based interface (VIZ). When selecting a specific report, its content is downloaded from the central repository file system to the browser. This system has been in use since 2004 at Hospital S. João. For access control this system uses a simple authentication and authorization procedure that is stored within a database and retrieves the user's profile (privileges and permissions associated with the role) each time he/she is successfully authenticated.

When genetic information was added to the repository, support for BTG became mandatory. There was the need to define the genetic group (so one more role) that had authorization to access genetic information while all the other users had to perform BTG in order to access the same information as they were not authorized to do it in normal circumstances.

The genetic group is comprised of 11 people. Table IV shows that in a 15 week period there were 86 authorized accesses to genetic information from 5 distinct users, while in 208 instances, 83 distinct members of staff needed to break the glass and gain access to that same information. In 177 instances, 98 distinct members of staff decided they did not have sufficient reason to break the glass. We know that in 156 instances the users answered no to the question of performing BTG while in 21 instances the users did not choose to answer the question and they probably just closed the browser or went to the previous window.

This EMR system has a total of 906 users and 3274 genetic reports stored within its repository (as of 26/08/2009).

TABLE IV – PRELIMINARY RESULTS FROM THE BTG IMPLEMENTATION FOR ACCESSES BETWEEN THE 13TH MAY AND THE 26TH OF AUGUST 2009.

	By authorized users (no need to BTG)	After agreeing to BTG disclaimer	Cancellations after BTG disclaimer
No of events	**86**	**208**	**177** (156 – The user said no to BTG) (21 – The user just closed the window)
No of distinct users	**5**	**83**	**98**

We can state that in this short period of time (only 15 weeks), and in order to enforce the legislation, we have already prevented 177 unauthorized accesses to genetic information. This more sensitive information was nevertheless openly available during all this time for those with genuine reasons to access it. Further, BTG can also be used to detect errors or mistakes within the policy as well as maintain data availability at all times, in a controlled and responsible way.

In the 208 instances where staff chose to break the glass, they had to state the reason for wanting to do this. They could either type in their own reason, which 67 people chose to do, or tick one of two preconfigured reasons. Table V presents the results. The first preconfigured reason is where staff members assert they are a member of a group who has access privileges, but for some reason they have not been granted access (37 people chose this reason). This is typically because of an administrative mistake where the user has not been assigned the correct role. The second preconfigured reason is where the users assert they should be granted access due to some emergency situation (104 people chose this reason). Remember that the user has been authenticated at this stage, and full audit logs are being recorded, so it is easy to identify which user actually broke the glass each time.

TABLE V – MOST COMMON REASONS GIVEN BY THE USERS WHO PERFORMED BTG.

Reasons to perform BTG	Total
I have urgency in seeing the requested information although I'm not normally allowed to do it	104
Write own reason	67
I should belong to the group that can access genetic information	37

The BTG concept implemented in the VEPR was done in a proprietary way. There is no BTG state information or the capability of a fine-grained definition of BTG in the initial implementation. This is why the implementation of the BTG-RBAC model in a state based RBAC engine described in this paper is now being implemented. It will help to enhance the use of BTG in our real setting and will provide for a more flexible and transparent way of controlling the need of users to access information in unanticipated situations, which they are not normally allowed to do.

V. CONCLUSION

This paper presents a new BTG-RBAC model that integrates BTG features within the NIST/ANSI RBAC model in an easy to use, secure and responsible way. The system is easy to use because the BTG-RBAC engine supplements the grant/deny response with an additional "permission to BTG" response. This allows applications to easily converse with the user and ask them if they would like to break the glass. We provide two alternative ways of specifying policy rules for BTG-RBAC policies, according to either the simple BTG-RBAC model or the complete BTG-RBAC model. The system is secure because it allows the administrator to add BTG in a controlled manner and the effects may be monitored closely through the provision of various obligations. The model allows users to act responsibly by giving them a choice whether to BTG or not, when they are initially denied access. The BTG-RBAC model can be implemented within any application and it provides for a more flexible, dynamic and adaptable access control policy that will relate more closely with end users' needs in complex settings.

ACKNOWLEDGMENT

The research leading to these results has received funding from the the (ISC)2 Organization and the Calouste Gulbenkian Portuguese Foundation, as well as the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 216287 (TAS³ - Trusted Architecture for Securely Shared Services)[1]. We would like to thank Dr Stijn Lievens for proof reading the mathematical expressions, and the anonymous ACSAC reviewers for providing useful feedback.

REFERENCES

[1] Anderson R. Security Engineering: A guide to build dependable distributed systems. Wiley 2001.

[2] Information Technology – Role Based Access Control. ANSI/INCITS 359-2004. International Committee for Information Technology Standards.

[3] Rissanen E, Firozabadi S, Sergot M. Towards a Mechanism for Discretionary Overriding of Access Control. Proceedings of the 12th International Workshop on Security Protocols, Cambridge. 2004.

[4] Povey D. Optimistic security: a new access control paradigm," in Proceedings of the 1999 workshop on New security paradigms. ACM Press, 2000; 40-45.

[5] Ferreira A, Cruz-Correia R, Antunes L, Farinha P, Oliveira-Palhares E, Chadwick D W, Costa-Pereira A: How to break access control in a controlled manner? Proceedings of the 19th IEEE Symposium on Computer-Based Medical Systems. 2006; 847-851.

[6] Break-glass: An approach to granting emergency access to healthcare systems. White paper, Joint –NEMA/COCIR/JIRA Security and Privacy Committee (SPC). 2004.

[7] Gansen Zhao, David Chadwick, and Sassa Otenko. Obligation for Role Based Access Control. *IEEE International Symposium on Security in Networks and Distributed Systems (SSNDS07)*. 2007.

[8] Break Glass – Granting Emergency Access to Critical ePHI Systems – HIPAA Security. Protecting the privacy and security of health information – HEALTH INSURANCE PORTABILITY & ACCOUNTABILITY ACT. Accessed at: http://hipaa.yale.edu/security/sysadmin/breakglass.html. Accessed on the 5th May 2009.

[9] 7869 2008-A SCR Clinical User Guide. NHS care records service – NHS Connecting for Health. 2008. Accessed at: http://www.connectingforhealth.nhs.uk/systemsandservices/nhscrs/scr/publications/7869_user_guide.pdf. Accessed on the 5th May 2009.

[10] McGraw R. Risk-Adaptable Access Control (RAdAC). Privilege (Access) Management Workshop. NIST – National Institute of Standards and Technology – Information Technology Laboratory. 2009.

[11] Rostad L. Access Control in Healthcare Application. NOKOBIT05. 2005.

[12] Rostad L., Edsberg O. A study of access control requirements for healthcare systems based on audit trails from access logs. Annual Computer Security Applications Conference (ACSAC). 2006.

[13] Brucker A., Petrisch H. Extending Access Control Models with Break-glass. SACMAT'09. 2009.

[14] Naftaly H., Lockman A. Ensuring integrity by adding obligations to privileges. Proceedings of the 8th international conference on software engineering. 1985.

[15] Jonscher D. Extending access controls with duties – realized by active mechanisms. Database Security IV: Status and Prospects. North-Holland. 1993.

[16] David W Chadwick, Linying Su, Romain Laborde. "Coordinating Access Control in Grid Services". Concurrency and Computation: Practice and Experience. 2008; Volume 20, Issue 9, Pages 1071-1094.

[17] Bettini C., Jajodia S., Wang X., Wijesekera D. Obligation monitoring in policy management. Proceedings of the 3rd international workshop on policies for distributed systems and networks. 2002.

[18] Bettini C.,Jajodia S.,Wang X., Wijesekera D. Provisions and obligations in policy management and security applications. In VLDB. 2002.

[19] Building a Modular Authorization Infrastrucutre. David Chadwick, Gansen Zhao, Sassa Otenko, Romain Laborde, Linying Su, Tuan Anh Nguyen. In All Hands Meeting, Nottingham. 2006.

[20] Cruz-Correia R, , Vieira-Marques P, Costa P, Ferreira A, Oliveira-Palhares E, Araújo F, Costa Pereira A. Integration of hospital data using agent technologies – a case study. AICommunications special issue of ECAI. 2005; 18(3): 191-200.

[21] Lei 12/2005 – Informação de genética pessoal e informação de saúde. Diário da República – Série A, nº 18. 2005.

[1] The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.