INCREASING PERFORMANCE AND GRANULARITY IN ROLE-BASED ACCESS CONTROL SYSTEMS

A CASE STUDY IN GRSECURITY

Bradley Spengler (bspengle@bucknell.edu)

ABSTRACT

There is a movement in progress among operating system vendors to bring greater security through preventative measures. One such measure making headway in the Linux operating system is Role-Based Access Control (RBAC). For such a measure to be widely accepted, it needs to have an insignificant performance hit and scale well in large corporate environments. Such a measure also needs to be intuitive and require as little manual configuration as possible, while at the same time providing an effective level of security. This paper provides a case study of an RBAC system and illustrates changes that were made in order to help it become more widely accepted and provide a strong level of protection while still being administrator-friendly. The changes involved altering data structures to remove algorithmically complex bottlenecks, providing dynamic unions of policy to ease configuration and reduce memory usage, allowing dynamic interpretation of wildcards in security policy, implementing an anticipatory memory allocator, and providing a robust method of defining allowed user and group transitions on a Linux system.

1. INTRODUCTION

In today's globally networked society, security is of great concern to everyone, from the individual home user to the corporate giants. Security is a concern for good reason, as the number of successful attacks against computer systems increases yearly. If a perfect solution in the technical sense to the problem of security existed, the global community would still not accept it. Often, the claim is made that performance must be sacrificed in the name of security. For this reason, many commercial entities are unwilling to add additional levels of security to their already resource-starved systems. In addition to a base level performance decrease, scalability plays a role, especially for large corporations who have thousands of users and incredibly complex and diverse software systems.

In addition to incurring a negligible performance hit, a security system that will be accepted by the global community must require little to no manual configuration. In other words, the security system must be a solution and not a toolbox for an administrator to hand-write custom policies for every machine under his/her control. Creating an effective security solution as opposed

to an effective security toolbox is perhaps not obviously a non-trivial task. A few questions that must be resolved are: How do we anticipate future legitimate usage? How do we determine what resources are important to protect? When do we decide to label an application as privileged? How do we reduce policies to a manageable size while retaining the same level of security?

Algorithms devised to solve the problems these questions describe are obviously complex in design and most likely complex in the time and space sense of algorithm complexity. Thus, innovative ideas need to be used for both the problems themselves and the solutions to the problems. The primary focus of my recent research has been to apply intelligent solutions to the already present solutions I have developed to answer the questions above.

A necessary decision in a usable security system is to abstract the policy definition from the details of the operating system. Administrators generally do not understand the security ramifications of every single system call implemented by the operating system, so by abstracting policy permissions into read, write, execute, etc. the administrator immediately knows what kind of operations are allowed on a resource. Even the most knowledgeable of administrators might not be aware that it is possible to kill a process not only with the kill system call, but also by using a number of flags of the fcntl system call.

Of course, there is a tradeoff being made here between granularity and usability. For a security system to be both flexible and tightly controlling of resources, it needs to provide a high level of granularity. One direction that can be taken in regards to this tradeoff is to only increase granularity where there is a definite security benefit. For instance, restricting whether a process can call the getpid system call has no security benefit, so enforcing such a thing in the security system would simply add bloat and be confusing to the administrator. Increasing granularity where there is a security benefit has been the secondary focus of my recent research.

2. GRSECURITY

The results of my research into performance and granularity in security systems have been implemented in the grsecurity [0] project. Grsecurity exists as a patch to the 2.4 and 2.6 trees of the Linux kernel and provides many security features [1] from the operating system level. It employs a "detection, prevention, and containment" [2] model through its wide spectrum of security enhancements.

Among its list of features include change root restrictions that disallow a process with a file system root changed by the chroot system call from performing certain operations that would allow it to access the global file system or affect outside processes in any way. Grsecurity also includes the work of the PaX project [3] that provides protection against exploitation of entire classes of address space bugs, including the most commonly exploited software vulnerability today: the buffer overflow. Additionally, grsecurity provides a robust Role-Based Access Control (RBAC) system that offers fine-grained auditing in addition to access control.

3. Role-Based Access Control (RBAC)

According to [4], Role-Based Access Control allows access decisions to be "based on the roles that individual users have as part of an organization." Roles are simply a descriptor for the kinds of tasks a user can perform. In the real world, a role can be a doctor, nurse, teller, manager, etc. In an RBAC system, a role can be created for a DNS server administrator, local users that only check their mail, or remote users of a source code repository like CVS. For these roles, the policy administrator can how users within those roles can interact with the system on a per-process basis. Essentially, each role will its own view of the file system and other processes. RBAC systems also carry over concepts from Mandatory Access Control (MAC) such as subjects and objects. Subjects are labels for processes, while objects are generally labels for files being accessed. [5]

Grsecurity takes the basic concept of a Role-Based Access Control system and provides several useful extensions.

In grsecurity, roles are split into three different categories: user, group, and special roles. User and group roles are roles that are tied to specific user or group IDs. To illustrate, if an application were to call setuid(31) and UID 31 had an associated user role, then that user role would be automatically applied to the application. Special roles however can be created and assigned arbitrarily as they have no association with the UNIX user or group ID system. Since a design goal of grsecurity is never to grant any privilege over what the Linux system would normally allow, special roles are incredibly useful for creating separate administrator roles on the system, as every administrator still needs to have UID 0.

Grsecurity also features IP-based roles. For any role, a list of IP addresses can be specified with optional netmasks that define which hosts can use the role. If a user logged in from a host not specified in the list for the role attempts to use the role, they will fall back through the role heirarchy until an applicable role can be found. In grsecurity, the role hierarchy moves from high to low in the following order: special role, user role, group role, default role. The default role is the role that is applied when no other role applies.

Grsecurity also implements role transition tables. These allow the policy administrator to specify the special roles to which a given role can transition. Grsecurity provides both authenticated and non-authenticated access to special roles. Role transition tables are necessary in the case of non-authenticated access and provide an additional layer of protection in the case of a compromised password for authenticated access.

Perhaps most importantly, grsecurity is the only free software security system that features an intelligent learning system that can generate least-privilege policies for the entire system with no configuration. The learning system intercepts every access checkpoint in the RBAC system and sends a log entry through a character device to an application in user space that caches the most frequently logged entries and writes all logs not existing in the cache to disk. Each log entry includes the full pathname of the binary or script that was executed for the process to be created, the UID and GID of the process, the full pathname, inode number, device number, and operation type of the target file, the number of the

Linux capability used, the socket type, protocol, IP address, and port involved in a socket operation, and the resource type and requested resource amount. With this information, very descriptive policies can be generated. The learning feature can be used on the entire system, a single role, or a single process. When learning is performed on a single role or process, the rest of the system can still be protected. Additionally, only access attempts that are denied for that role or process are logged by the learning system, so it can be very useful in correcting a non-functioning policy.

The learning log analyzer and rule set generator is a very complex program as it attempts to solve the difficult problems stated in the introduction. The application performs a complex multi-stage graph and heuristics-based reduction and analysis of rules. Reduction involves reducing the number rules in a policy in appropriate ways. For instance, reducing 1000 similar accesses to randomly named temporary files located within /tmp is appropriate, while reducing two different accesses to important files within /etc is not. Analysis involves detecting whether a reduction led to a weakened policy and auto-correcting the weakness. For instance, if a number read accesses in /etc were reduced to read access to all of /etc, this would cause the problem of leaving the private SSH keys located in /etc/ssh readable. In this case, the analysis algorithm would insert a rule disallowing reads to /etc/ssh. Because these processes are so complex in terms of time and memory consumption, the performance review accomplished through my research was most welcome.

4. METHODOLOGY

To identify the performance bottlenecks in gradm, the RBAC administration utility of grsecurity, I compiled the utility with profiling support. So that any function of linear or polynomial complexity would be clearly visible, I created a rule set with over 300,000 objects. The following is the resulting profile of the application:

% time	cumulative seconds	self seconds	self calls	self s/call	total s/call	name
53.89	35.84	35.84	341732	0.00	0.00	is_proc_object_dupe
43.45	64.73	28.89	49	0.59	0.59	expand_acl
1.34	65.62	0.89	405541	0.00	0.00	gradmlex
0.59	66.01	0.39	1	0.39	37.45	gradmparse
0.38	66.26	0.25	325745	0.00	0.00	add_proc_object_acl
0.14	66.35	0.09	379	0.00	0.00	check_permission
0.09	66.41	0.06	1	0.06	0.06	conv_user_to_kernel

The important area to note here is the leftmost column, which identifies the percentage of time spent in a single function out of the entire application. It is clearly visible that there are two huge bottlenecks: is_proc_object_dupe and expand_acl. Upon analysis of the source code of gradm, it was determined that the is_proc_object_dupe function has O (n) complexity, where n is the number of objects within the subject currently being parsed from the centralized policy file. However, because the function is called during the addition of every object, the total effective complexity of using the function becomes $O(n^2)$.

The second function, expand_acl, is related to subject inheritance in the centralized policy file. In grsecurity's RBAC system, the configuration language supports inheriting the policy of a parent subject, where parent subject is defined as a subject placed on a parent directory of the inheriting subject. To illustrate using a sample configuration:

```
subject /
        /               r
        /opt            rx
        /home           rwxcd
        /mnt            rw
        /dev
        /dev/grsec      h
        /dev/urandom    r
        /dev/random     r
        /dev/zero       rw
        /dev/input      rw
        /dev/psaux      rw
        /dev/null       rw
        /dev/tty?       rw
        /dev/console    rw
        /dev/tty        rw
        /dev/pts        rw
        /dev/ptmx       rw
        /dev/dsp        rw
        /dev/mixer      rw
        /dev/initctl    rw
        /dev/mem        h
        /dev/kmem       h
        /dev/port       h
        /bin            rx
        /sbin           rx
        /lib            rx
        /usr            rx
        /etc            rx
        /proc           rwx
        /proc/kcore     h
        /proc/sys       r
        /root           r
        /tmp            rwcd
        /var            rwxcd
        /var/tmp        rwcd
        /var/log        r
        /boot           r
        /etc/grsech
        /etc/ssh        h
```

```
        -CAP_SYS_TTY_CONFIG
        -CAP_LINUX_IMMUTABLE
        -CAP_NET_RAW
        -CAP_MKNOD
        -CAP_SYS_ADMIN
        -CAP_SYS_RAWIO
        -CAP_SYS_MODULE
        -CAP_SYS_PTRACE
        -CAP_NET_ADMIN
        -CAP_NET_BIND_SERVICE
        -CAP_SYS_CHROOT

subject /usr/sbin/sshd  dp
        /etc/ssh        r
        /dev/log        rw

        +CAP_SYS_TTY_CONFIG
        +CAP_SYS_CHROOT
```

The important thing to note in this configuration is that the administrator only needed to add two more Linux capabilities and more privileged access to two files over the parent subject. To simplify configuration and improve policy readability, he/she used configuration inheritance to make the /usr/sbin/sshd subject inherit the subject of / in this case, as / was the closest matching parent directory. To reduce complexity of policy enforcement in the kernel, it was initially decided to expand the inheritance within the administration utility itself. The expand_acl function performs this static subject inheritance. To accomplish this, the expand_acl function iterates upon every object in the parent subject and compares the object's filename with the filename of every filename in the inheriting subject. If the filename does not exist in the inheriting subject, then the object in the parent subject is copied into the inheriting subject. Clearly, this function has polynomial complexity due to its nested loops. It is specifically O(m*n), where m is the number of objects in the parent subject and n is the number of objects in the inheriting subject.

Other bottlenecks were identified without the need for profiling. To reduce the kernel complexity of access lookups, a decision was initially made to have the administration utility expand the '*' and '?' wildcard characters in objects. This has a number of effects on both performance and granularity. Since the wildcards are interpreted at policy enable time, an object like /tmp/* does not apply to temporary files created after the policy is enabled. Additionally, every existing file matching the wildcarded has to be accessed to obtain its inode and device numbers, which increases run time proportional to the number of files the object expands to. This also has an effect on performance in other areas of the code as the additional objects further demonstrate the high algorithmic complexity of certain functions, such as the two analyzed earlier. For example, an object like /dev/* will on most Linux systems expand to over 1500 different objects. Accessing each of these files and creating new object structures for each

entry takes a considerable amount of time relative to the total run time of an otherwise small rule set.

The learning log analysis and rule set generation code utilizes a graph structure to perform rule set reducation. A level of abstraction is provided so that the same data structure can be used when performing the heuristics-based reduction. Though a discussion of the actual algorithms behind this reduction is not appropriate for this paper, what is of importance is how the process of constructing the graph structures affects performance. Each vertex in the graph contains a dynamically allocated array of pointers to adjacent vertices. Upon addition of another adjacent vertex during graph construction, the array containing the pointers to adjacent vertices is resized to a size equal to the current size plus the size necessary for the storage of an additional pointer. Additionally, because the graphs that describe the access profile parsed from the learning log are generated immediately, there are many calls to the dynamic memory allocator to store the information for each vertex. The resizing of pointer arrays can cause poor performance, as the reallocation function may have to move the array to resize it. When dealing with large numbers of dynamic memory allocations, the brk system call is frequently called to expand the size of the heap. The resulting transfer of control to the kernel on a constant basis greatly decreases performance.

One area that I noticed needed improvement in terms of granularity involved the superuser's ability to transition to other users or groups. This can be done through the family of system calls that allow a privileged user to modify the current process' effective, real, saved, or file system user or group IDs. It can also be accomplished by executing a setuid or setgid application with the owner or group of the binary set to the user or group the user wishes to transition to.

On an unmodified Linux system, superusers have the ability to bypass the normal DAC system, effectively allowing them to modify files owned by other users, even if there are no such permissions specified that would normally allow such a modification. Linux does provide, however, a capability named CAP_DAC_OVERRIDE that allows one to disable the ability to override the DAC system for a given process. Linux also provides two capabilities named CAP_SETUID and CAP_SETGID that permit a process to transition to other users or groups. In the presence of these capabilities, a process can effectively recreate the CAP_DAC_OVERRIDE capability by first checking the ownership of a file it wishes to access and then transitioning to the owner of the file. After it accesses the file, the process can return to its old user or group ID and repeat the process for another file. In RBAC it is important to interlock with the DAC permissions by adding policy enforcement at areas in which a process can modify its state to conform to the DAC permissions and be able to modify a given file. There is a clear lack of granularity in this case that has an additional effect. In grsecurity's RBAC system, changing one's real UID or GID causes an immediate transition to an applicable user or group role. It is important for these transitions to be protected as well, since a process with CAP_SETUID or CAP_SETGID can arbitrarily change to any user or group role.

5. IMPLEMENTATION

In the previous section, a profile of the administration utility was presented that demonstrated two functions with high algorithmic complexity. The two functions, is_proc_object_dupe and expand_acl constitute over 96% of the running time of the application. To correct these two bottlenecks, I proposed two changes in my research: a hash table data structure for each subject that contains pointers to each object belonging to the subject, and kernel interpretation of configuration inheritance.

Three goals were made for the conversion to hash tables. The objects belonging to each subject should still be able to be traversed in a list fashion, not by skipping over null entries in the hash table. In addition, the change should be abstracted from higher-level functions and thus require minimal changes to the high-level code. The hash tables should additionally be searchable by filename.

The hash function used is the same provided by the Linux kernel to hash filenames. Quadratic probing was chosen to handle collisions in the hash table due to its simplicity and effectiveness in preventing primary clusters. Because objects would be continually added to the subject and the number of them would not be pre-determined, a method to resize the hash table was created. It was noted that although the number of subjects in a given policy is much smaller than the number of objects, the same $O(n^2)$ algorithm applied to duplicate checking of subjects. For this reason, similar hash tables were created to store pointers to each subject belonging to a given role. By making these changes, several expensive routines in the administration utility were changed to constant time complexity.

Below is a profile taken of the application after the algorithmic enhancements. You will notice that the previous two bottlenecks have disappeared and that now the lexer is the most time intensive operation of the application. You will also notice the time percentages are more evenly distributed among the functions, unlike previously where two functions alone made up 96% of the running time.

% time	cumulative seconds	seconds	self calls	self s/call	total s/call	name
18.94	0.32	0.32	405541	0.00	0.00	gradmlex
14.20	0.56	0.24	24992880	0.00	0.00	partial_name_hash
8.88	0.71	0.15	352031	0.00	0.00	lookup_name_entry
8.28	0.85	0.14	1088522	0.00	0.00	insert_name_entry
7.69	0.98	0.13	336152	0.00	0.00	lookup_hash_entry
7.10	1.10	0.12	1440541	0.00	0.00	full_name_hash
6.51	1.21	0.11	712337	0.00	0.00	insert_hash_entry
6.51	1.32	0.11	1	0.11	1.62	gradmparse
5.92	1.42	0.10	1440541	0.00	0.00	nhash
4.14	1.49	0.07	1	0.07	0.07	conv_user_to_kernel
3.55	1.55	0.06	325745	0.00	0.00	proc_object_mode_conv
3.25	1.61	0.06	1048477	0.00	0.00	fhash
2.96	1.66	0.05	325745	0.00	0.00	add_proc_object_acl

0.59	1.67	0.01	351346	0.00	0.00	is_proc_object_dupe
0.59	1.68	0.01	336077	0.00	0.00	insert_acl_object
0.59	1.69	0.01	11058	0.00	0.00	is_deleted_file_dupe

The change to a kernel interpretation of configuration inheritance required many changes. On the user land end, a pointer had to be added to each subject, which referenced its parent subject. Until all rules from the configuration file are parsed and converted into usable data structures, the pointer remains null. After this time, each subject of every role is checked to see if it is using configuration inheritance. If it is, each trailing path components of the current subject's filename is stripped off and a hash lookup is performed to find a subject in the current role with that modified filename. This process iterates until the remaining filename is the file system root. When a matching subject is found, the parent subject pointer for the current subject is set to the matching subject. If the subject does not use configuration inheritance, the parent subject pointer remains null. What essentially results when this entire process is complete are chains of inheritance for subjects. This happens because a subject that is inherited by another subject can itself inherit a subject, and so on. The chains are traversable by following the parent subject recursively until a null parent subject pointer is found.

The kernel land changes required for the new configuration inheritance system were much more extensive. The process of passing the generated policy to the kernel for enforcement essentially involves the kernel taking a pointer from user space and copying the structures from user space directly, following any pointers located within the structures and copying them as well. Before the new configuration inheritance system, it was never the case that two pointers in the information passed to the kernel referenced the same location. Now with the new system, a method had to be implemented to determine if a subject had already been copied to the kernel, so that the parent subject pointer could reference the kernel address of the already copied parent subject. Since the number of subjects being copied to the kernel is pre-determined, a hash table was created that mapped the user space address of a subject to its address in the kernel. Upon seeing a parent subject pointer, its user space address is looked up in the hash table. If a match is not found, then the subject is copied into kernel memory, the hash table is updated, and the address of the subject in kernel memory is returned. Otherwise, the matching kernel address is returned. The caller of the function sets this address to the parent subject pointer. The low-level access check functions also had to be modified for the new inheritance system. Normally, upon attempted access of a file, the RBAC system would traverse down the file system tree towards the real file system root, checking every path along the way for a matching object in the subject of the current process. If a matching object were found, then the requested permissions would be checked against the granted permissions, and either success or failure would be returned in the appropriate cases. With the new configuration system, at every iteration of traversing down the file system tree, the chain of parent subjects is followed, and a lookup for the path at the current iteration is performed to attempt to obtain an object. If an object is found,

then the lookup function proceeds as before with checking permissions and returning success or failure.

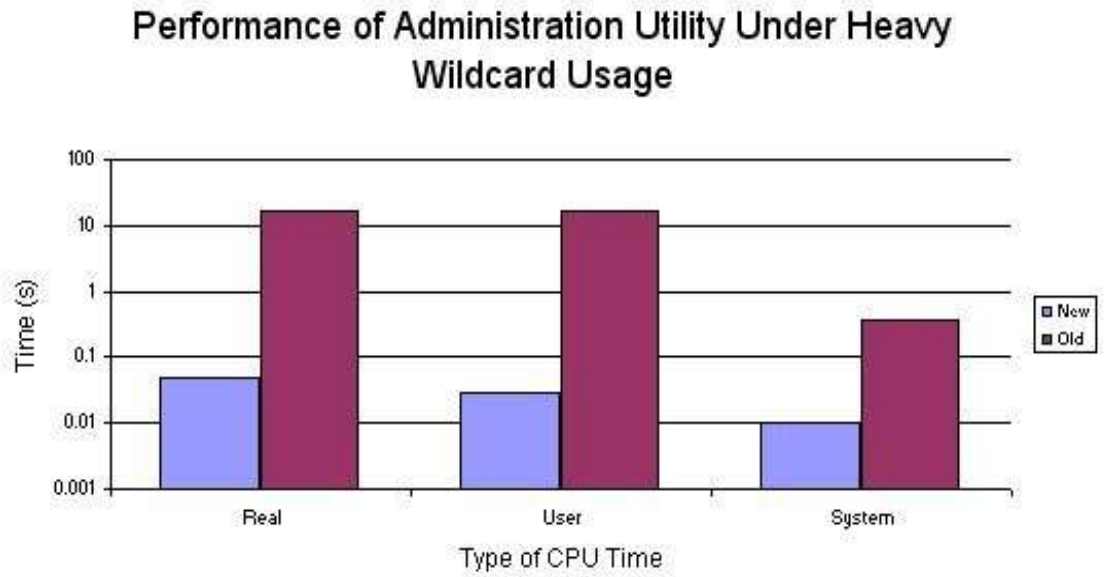
Not only does this change to a kernel interpretation of configuration inheritance result in performance improvements in user space, but it also results in huge memory savings in the kernel. The memory savings are most dramatic when the inherited subject has many objects and there are many subjects inheriting that subject. Upon a poll of users of the RBAC system, it was shown that many use this form of configuration and will thus benefit greatly from such a change.

The addition of a kernel interpretation of wildcard characters in objects was a welcome change among users. It provides a dynamic matching of the object containing wildcards to the file being accessed. This results in large memory savings in the kernel in many instances. For the feature to be viable performance-wise, however, it had to have no impact on the performance of normal access lookups. With this in mind, a suitable solution was developed. The general idea was to attach an array of filenames and allowed permissions to an object, and when the access lookup matched that object when no explicit object existed for the file being accessed, the filenames in the array would be checked against that of the file being accessed.

To accomplish this, two problems had to be solved. First, we needed to decide what object the wildcarded object would be attached to. Second, we don't want to generate a filename for every file access, as the RBAC's access lookups uses the structures provided by Linux's Virtual File System (VFS) to identify files by inode and device number. The solution to the first problem was to find the first wildcard character in the wildcarded filename, and truncate the filename at the point where the path component started that contained the wildcard character. So for example, if the object were /home/*/test/*/file, then the object that the wildcarded object would be attached to would be /home. We can be assured in the kernel that this is the case since the administration utility will enforce that there be an object for /home when it finds a wildcarded object that it needs to attach to the /home object. This solution also implies that the access lookup functions will favor non-wildcarded objects. That is, if the objects /home/*/test/*/file and /home/user1/test exist and the file /home/user1/test/dir1/file is accessed, the /home/user1/test object will match first and thus be used to check permissions and grant or deny access to the file. To solve the second problem, filenames are generated by demand by the access lookup functions in the RBAC system.
Additionally, so that the filename is not recreated every time a wildcarded object is encountered in a lookup for a single file, the lower level access lookup functions have as a parameter a pointer to a pointer on the stack that references the created filename. If the filename has not been generated yet, the dereferenced pointer results in a null pointer, at which point the code will generate the filename and update the pointer to reference the generated filename. If the dereferenced pointer is non-null, then the resulting pointer is used as a reference to the filename. Currently, only the '*' and '?' wildcards are expanded, however this could easily be extended to support such objects as /dev/tty[0-9]+.

The following chart demonstrates the performance benefit of the kernel

interpretation of wildcarded objects. It represents the time required to enable a policy through the administration utility. The rule set that was used contained a combination of 4,000 objects and wildcarded objects. With the kernel interpretation of wildcarded objects, these objects are not expanded at all, and only 4,000 objects are sent to the kernel. Without this new enhancement, the wildcarded objects would expand to over 400,000 objects that would all be passed directly to the kernel. You can see an enormous difference in running times of the two versions.

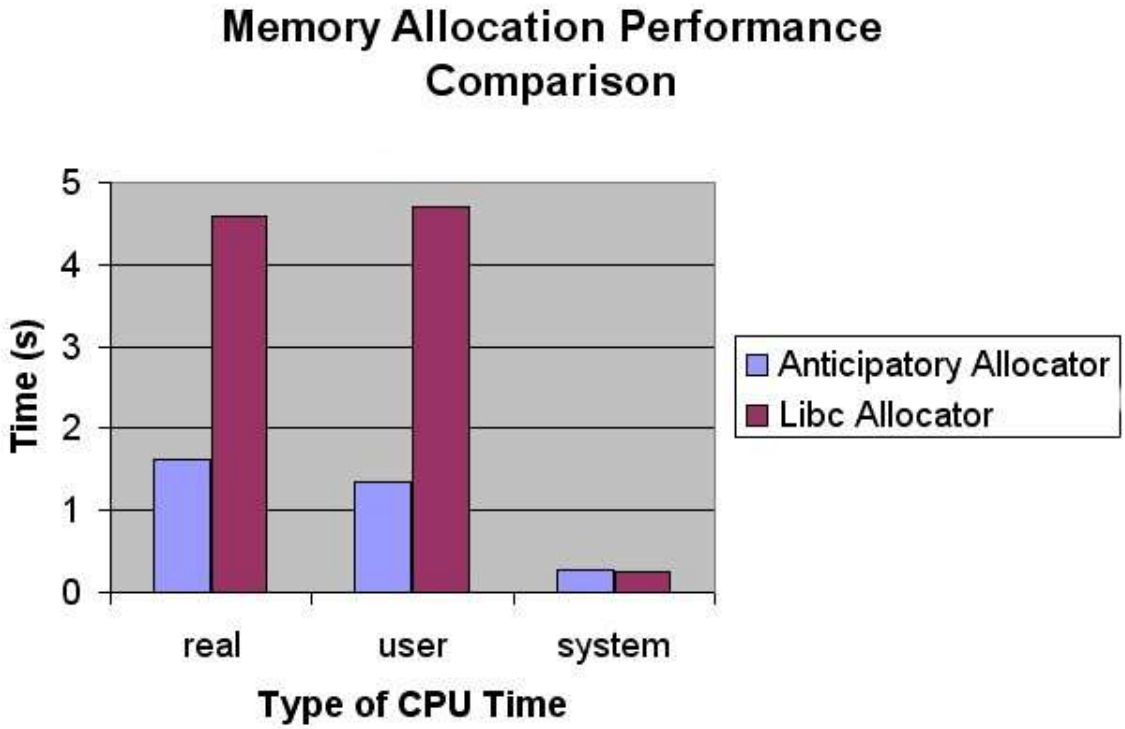


To solve the problems regarding dynamic memory allocation and resizing in the learning log analysis and rule set generation code, an anticipatory memory allocation system was developed. The allocation system has two parts: an allocator and a reallocator. The allocator allocates a number of structures that hold pointers to large contiguous buffers. These buffers are used for memory allocations that are not meant to be resized. Initially, the pointers to the contiguous buffers are null. When an allocation request is made, the algorithm checks each of the structures to see if its pointer to an allocation buffer is not null. If the pointer is not null, the algorithm checks to see if there is enough room in the contiguous buffer for the allocation, and returns the allocated address if so. If there is not enough room, the algorithm continues through all the structures looking for a contiguous buffer with enough room, or a null pointer. If a null pointer is found, then a contiguous buffer is allocated and the address returned is the start of the buffer. If the algorithm iterates to the last structure with no success, then additional structures are added and a new contiguous buffer is allocated to service the current allocation and allow for future allocations. Because of the usage of memory allocations in the specific instance of the learning log analysis and rule set generation code, a method to free allocated memory was not needed. All allocated memory is generally in constant use until the process exits.

The reallocator requires a special allocator that allows for reallocation. When an allocation request is made, a

buffer is allocated that allows room for a structure that contains information about the current usage of the buffer and its maximum size. The allocated buffer also allows for a large gap at the end for future growth. The returned address is the start of the buffer plus the size of the allocation structure. Upon a reallocation request, the address passed to the function is subtracted by the size of the allocation structure, and then its members are checked to see if there is enough room in the current buffer to satisfy the request. If there is, then the address passed in is simply returned. If there is not, then the reallocation function provided by libc is called with a size large enough to service the current request and several future requests. The address returned is the start of the reallocated buffer plus the size of the allocation structure. To free these kinds of allocations, the free function simply subtracts the size of the allocation structure and frees the resulting pointer.

To test the performance of the new memory allocation functions, an application was written that simulates the usage of memory allocation in the learning log analysis and rule set generation code. The following chart demonstrates the significant improvement over the standard memory allocation functions. The only unusual difference between the two running times is the system time. The small difference, around .03s, can be explained for. When large contiguous memory allocations are made, the libc allocation functions, which the new allocation functions wrap around, do not use the brk system call. Instead, they create an anonymous mapping through the mmap system call. The mmap system call is more complex than the brk system call, thus resulting in the difference in system times.



Finally, to improve the granularity of security checks around transitioning to different users or groups, I have added the following syntax to the RBAC system's configuration language:

```
user_transitions_allow user1 otheruser anotheruser
group_transitions_allow group1 othergroup anothergroup

user_transitions_deny root secretuser
group_transitions_deny root secretgroup
```

Notice that the ability is given to control access to users or groups on an inclusive or exclusive basis. To enforce this, checks were added to every location in the kernel where a process' user or group credentials change, namely the set*id family of functions and the routines that handle setuid and setgid executables. If a transition to a user or group is not allowed, then a process' effective, real, saved, or file system user or group ID cannot be changed to the ID for that user or group.

6. CONCLUSION

In summary, these performance and granularity changes to grsecurity's RBAC system have greatly increased its ability to be a usable security system and not just a tool box. The methodology applied to this project to increase performance and scalability can be used towards other projects as well. Thanks to these improvements, grsecurity's RBAC system can now scale well in large corporate environments, an area where many other systems would fail.

7. FUTURE WORK

In the areas of performance and granularity, there is always room for improvement. In light of this, I would like to expand upon my current research with several projects to improve security while requiring little configuration, improve the performance of grsecurity's learning system in disk, memory, and CPU usage, and improve granularity in important cases where a full system compromise can be prevented.

First, I would like to transpose an integrity model on top of the current RBAC framework of grsecurity. An integrity model works well to prevent untrusted data from affecting trusted services. A great benefit of an integrity model is that it requires little if any manual configuration. The LOMAC [6] project will most likely serve as a guide for this work.

Second, I would like to develop a transparent method of sandboxing applications. The proposed mechanism is as follows: a process under the control of the sandbox will have its file system calls intercepted such that for any operation that would modify the file system, a private copy of that file is made that is operated upon. When the process exits, the modified files are stored in a compressed archive and associated with the process. If the application executes another program, the new program will be sandboxed as well. This has an advantage over the chroot system call as it does not require any source modification or the duplication of lots of read-only code and data. The sandboxing will be able to be turned on for a per-role and per-process basis in the RBAC system. An example

use of this would be to sandbox all users' shells. No files they create will exist on the real filesystem, such as all files in their home directories. Additionally, this mechanism removes an entire class of bugs involving insecure file permissions, as any files with improper permissions will not be accessible by other, possibly malicious, users. Exploiting a setuid root application on the system will not allow them to modify /etc/passwd or /etc/shadow, for instance, since the change will only be made to their private copies of the files. Note however, that sandboxing in this manner will not solve the problem of a user being able to read the contents of /etc/shadow, cracking the password, and logging in as root or another user. For this reason, the sandboxing will be integrated into the RBAC system.

Additionally, I would like to implement a similar feature to one planned by the Linux kernel developers: role-based resource limits. Currently, all resources limits except for number of processes are enforced per-process. Limiting of number of processes is done on a per-user basis. Linux kernel developers had planned to extend some of the other resource limits to be per-user as well, however no work has yet been done on this. I would like to extend the resource limits to apply to an entire role, where such an extension is reasonable. Resource limits such as maximum file size would be acceptable candidates for such an extension.

Granularity in the case of applications such as XFree86 is very important. XFree86 uses privileged I/O ports on the Intel Architecture and accesses physical memory directly through a character device provided by Linux. Access to privileged I/O ports is very coarse-grained on Linux and other UNIX-like operating systems. Only access to the first 1024 I/O ports are allowed to be specifically configured through the ioperm system call. To allow access to any of the remaining 64512 I/O ports, the iopl system call must be used. By using the iopl system call to enable such access, immediately all 65536 ports are allowed to be accessed. I plan to work with the PaX project on extending per-port configuration to all 65536 I/O ports. As for direct physical memory access, I plan on allowing to specify within the RBAC configuration language, which ranges within a file or device can be accessed for reading or writing. These two changes alone greatly reduce the previously massive security risk of running XFree86 or other similarly complex applications on trusted systems.

To ease configuration and give an additional safety blanket to all applications on the system, I propose to add a global role to the RBAC system. Upon every access lookup within the RBAC system, the global role will be consulted first, followed by the role of the current process. From a usability standpoint, the global role will give the administrator instant information about what kind of accesses will be impossible on the system, no matter how the rest of the policy is configured. It will also reduce the amount of policy duplicated among roles by having it stored in one central location.

Though there is little left to be done in the kernel portion of grsecurity's RBAC system in terms of performance, I plan on adding caching to access lookups. The reason for caching is that though access lookups are generally O(1), they can be as bad as O(m*n) where m is the inheritance depth of the current

subject and n is the number of path components of the pathname being accessed. Though in practice both of these numbers are very small and unlikely to cause any scalability issues, caching will aid in reducing the lookup cost of frequent accesses to the same file. The cache will map an inode and device pair to the object that would result if a full lookup were performed. The cache will be invalidated upon creation, deletion, and renaming.

Performance in the learning system of grsecurity is currently a great concern. Many users are generating learning logs that are hundreds of megabytes large. Attempts to analyze the learning logs and generate a working policy out of them result in out of memory conditions. To alleviate these problems, I plan on taking several approaches. First, I would like to change the learning reduction code so that it is done in multiple passes. That is, the logs will be read multiple times, but the generated policy will not be created and written out all at once. The policy may be generated one role or process at a time, where only the data relating to that role or process will be in memory at one time. Second, I would like to perform simple rule reduction before learning logs are written to disk by the learning daemon. In cases of applications like top, where pathnames matching the regular expression /proc/*/exe are continually accessed, I would like to reduce those accesses to the regular expression before being written to disk. This will result in huge disk savings. I would also like to perform reduction at the stage before access information is inserted into a graph structure. At that stage, all accesses are sorted by filename, which allows one to easily identify if there have been many accesses of the same kind to a single directory. By reducing these accesses before they are inserted into the graph structure, we will greatly reduce the amount of time required to analyze and generate policies for a heavily used system.

REFERENCES

[0] Grsecurity homepage. http://www.grsecurity.net

[1] Grsecurity features page. http://www.grsecurity.net/features.php

[2] Brad Spengler. Detection, Prevention, and Containment: A Study of Grsecurity. http://www.grsecurity.net/grsecurity-slide_files/frame.htm

[3] PaX homepage. http://pax.grsecurity.net

[4] NIST/ITL Bulletin, An Introduction to Role-Based Access Control. December, 1995. http://csrc.nist.gov/rbac/NIST-ITL-RBAC-bulletin.html

[5] Ravi S. Sandhu. Lattice-Based Access Control Models. ACM Journals Nov 2003. vol 26, issue 11, p9-19. http://www.list.gmu.edu/journals/computer/i93lbacm(org).pdf

[6] Timothy Fraser. "LOMAC: MAC You Can Live With," in the Proceedings of the FREENIX Track, 2001 USENIX Technical Conference, Boston, Massachusetts, USA, 2001. http://opensource.nailabs.com/lomac/docs/lomac-freenix01.pdf