Capabiilty-based Egress Network Access Control by using DNS Server

Shinichi Suzuki [a], Yasushi Shinjo [a,*,1], Toshio Hirotsu [b,1], Kozo Itano [a,1], Kazuhiko Kato [a,1]

[a] *Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki, 305-8577, Japan*

[b] *Department of Information and Computer Sciences, Toyohashi University of Technology, Toyohashi, Aichi, 441-8580, Japan*

Abstract

In conventional egress network access control (NAC) based on access control lists (ACLs), modifying the ACLs is a heavy task for administrators. To enable configuration without a large amount of administrators' effort, we introduce capabilities to egress NAC. In our method, a user can transfer his/her access rights (capabilities) to other persons without asking administrators. To realize our method, we use a DNS cache server and a router. A resolver of the client sends the user name, domain name, and service name to the DNS cache server. The DNS server issues capabilities according to a policy and sends them to the client. The client puts these capabilities into the IP options of packets and sends them to the router. The router verifies the capabilities, and determines whether to pass or block the packets. In this paper, we describe the design and implementation of our method in detail. Experimental results show that our method does not reduce the router's performance.

Key words: Access control, Capabilities, Network security, Egress filters, DNS

1 Introduction

Access control is one of the most important issues in computer security. In recent years, it has been performed not only at end hosts but also at inter-

* Corresponding author.
Email address: `yas@cs.tsukuba.ac.jp` (Yasushi Shinjo).

[1] CREST, Japan Science and Technology Agency (JST)

mediate routers and proxies. We call the access control that is enforced by network administrators at routers or proxies *network access control* (NAC).

NAC can be classified into two types: ingress NAC and egress NAC. Ingress NAC protects internal servers from external attackers and intruders. Egress NAC prevents internal users accessing undesired or dangerous external servers. Egress NAC is increasingly attracting the attention of educational organizations, government organizations, and companies.

Most implementations of conventional egress NAC use filters in routers or proxies. In such egress filters, network administrators describe a filtering policy based on *access control lists* (ACLs). A typical ACL is a list of filtering rules. Each rule consists of a user name, domain name, service name, and access permission or refusal. We call such egress NAC *ACL-based egress NAC*.

In ACL-based egress NAC, only network administrators can modify ACLs. Changing access rights needs to modify the ACLs, so normal users cannot transfer their access rights to other persons. For example, consider a section chief and a subordinate working together on a project in a company. The chief has the right to access an external site, but the subordinate does not. The chief can show the subordinate the external site by using his/her own computer. If the chief is willing to allow the subordinate to access the external site temporarily, he/she must request a network administrator to modify the ACL. The network administrator must modify the ACL to satisfy the request and restore it after the temporary access ends. Changing such sensitive information increases the workload of the network administrator.

To reduce the network administrator's burden, we introduce capabilities into egress NAC[1]. We call such an egress NAC *capability-based egress NAC*. In our method, a capability is a binary value that acts as an access key allowing the holder to access a specified server. Capabilities can be transferred by various methods, such as e-mail, files, and instant messages. Using capabilities has the advantage that a user can transfer his/her own access rights to other users. In capability-based egress NAC, a network administrator is freed from having to change ACLs.

In our method, we use DNS cache servers and routers together. Most client processes use a DNS cache server to resolve a name before accessing an external server. Therefore, the DNS cache server is one of the best places for issuing capabilities. The routers verify the capabilities, and passes or blocks packets.

Since capabilities could be stolen, to invalidate stolen capabilities, we issue capabilities dynamically and attach a time to live (TTL) to each one.

To realize our method, we have modified a library in Linux, the Linux kernel, a DNS cache server, and a router. We have also implemented utility programs that manage capabilities.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 explains our capability-based egress NAC. Section 4 presents experimental results. Finally, section 5 concludes this paper.

2 Related work

The Platform for Internet Content Selection (PICS)[2] enables labels (metadata) to be associated with Internet contents. PICS is designed to help parents and teachers control what children can access on the Internet. It also facilitates other uses for labels, including code signing and privacy. Based on the PICS platform, filtering software has been built. Some filtering products[3][4] can interpret HTTP, SMTP, NNTP, etc. with proxies. Since our method replaces packet filters, it can be used together with such proxies.

Authentication Gateway[5], Opengate[6] and Service Selection Gateway[7] are captive portals that perform egress NAC. A captive portal redirects all Web requests to a built-in Web server until a user has been authenticated by the Web server. When the Web server authenticates the user, the router changes the filtering rules to pass the user's packets. Since those captive portals are also based on ACL, changing access rights involves network administrator effort.

SOCKS[8][9] is a proxy protocol on transport layer. It can authenticate the user for each connection. The current NAC of SOCKS is based on ACLs. We could make capability-based egress NAC by using SOCKS, but we use the IP option, so we choose not to use SOCKS.

Amoeba[10] is a distributed operating system. It protects files and directories based on capabilities. In Amoeba, a directory server (name server) translates an ASCII file name into a capability. To access the file, processes need the capability. We refine directory servers for egress NAC.

3 Capability-based egress NAC

3.1 Environment

Fig. 1 shows our target environment. The internal network is separated from the external network by a router. The internal network has a DNS cache server. The router has a packet filter. Users login to client hosts on the internal

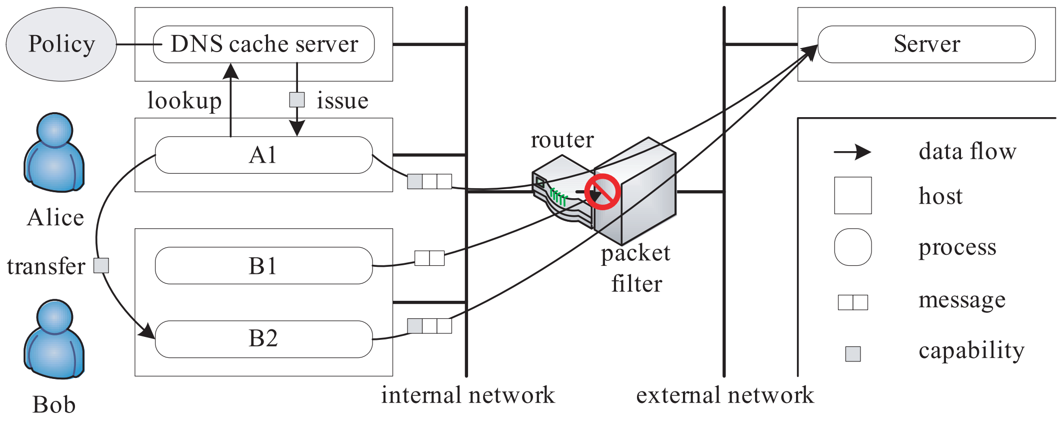


Fig. 1. Overview of capability-based egress NAC with client, DNS server, and router.

network. A user can login to multiple hosts at the same time. Multiple users can share a single client host at the same time. External server processes work on external server hosts. Multiple external server processes can work on a single external server host at the same time.

3.2 Overview of capability-based egress NAC

To make capability-based egress NAC, we use a DNS cache server and a router together. Fig. 1 shows an overview of our method.

When a user accesses an external server, he/she runs a client process. The process attaches the user name and service name to a DNS query message, and sends the message to the DNS cache server. The DNS cache server issues a capability according to the policy and sends back a DNS answer message with the capability. When the process tries to send a packet to the external server, the process attaches the capability to the packet and sends it to the router. The router verifies the capability of the packet by using a packet filter. If the capability is correct, the router passes the packet. If the packet does not include a capability or if the capability is incorrect, the router blocks the packet.

A user can transfer capabilities to another user. In Fig. 1, Alice has the right to access an external server. When a client process (A1 in Fig. 1) tries to access the external server, a capability is issued for the process. When the process sends a packet with the capability, the packet can go through the router and reach the server. Bob does not have access rights to the external server and a client process (B1 in Fig. 1) does not have a capability for the server. When the process sends a packet without a capability, the packet cannot go through the router. Alice can transfer the capability for the server from her process to Bob's process (B2 in Fig. 1). Now, if his process sends a packet with the capability, the packet can go through the router and reach the server.

In the paper [1], we have described a method that uses an operating system kernel to manage capabilities. In this paper, we use a utility program called *capability-agent* to manage capabilities outside the kernel. This utility program is designed based on ssh-agent[11]. Ssh-agent is a program that holds private keys for user authentication with public keys. Like ssh-agent, capability-agent communicates with applications through UNIX domain sockets.

Since capabilities can be stolen, to invalidate stolen capabilities, we give each capability a time to live (TTL). Therefore, a stolen capability expires and becomes ineffective after this time.

3.3 Effective usage of capabilities and utility programs

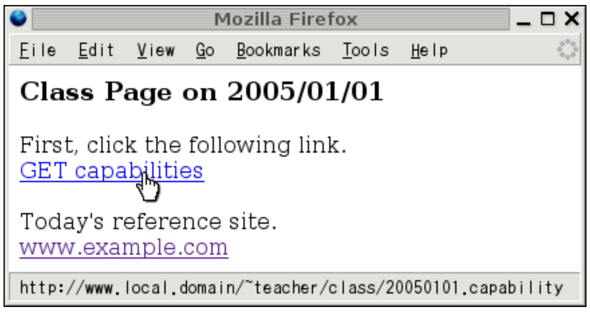
In Section 1, we have shown the effective usage of capabilities in a company. In this section, we show another effective usage of capabilities in an educational organization.

A teacher would often like to show school children a particular external web site that is blocked for them. Capability-based egress NAC is useful in this situation. Table 1 shows our utility programs. First, the teacher runs the utility program *capability-exp* to get the capability for the site and saves the capability into a file. Second, the teacher uploads the file to a web page for the children (Fig. 2). Third, the children access the page and download the file. The browser executes the utility program *capability-ctl* according to the MIME-type (application/capability-ctl) and the extension (.capability) of the file. This program reads the capability from the file, and imports the capability to his/her capability-agent. Finally, the children can access the external site with the browser.

The capability in the file is temporary. When the capability expires, the school children can no longer access the site. In capability-based egress NAC, none of the processes requires a network administrator. In ACL-based egress NAC,

Table 1
Utility programs.

Name	Description
capability-agent	Holds a capability list.
capability-ctl	Manipulates a capability list in capability-agent.
capability-exp	Exports capabilities from capability-agent and stores them into a file.



Class Page on 2005/01/01

First, click the following link.
GET capabilities

Today's reference site.
www.example.com

http://www.local.domain/~teacher/class/20050101.capability

Fig. 2. Screenshot of class page

the teacher must request a network administrator to change the ACL before the class. Furthermore, the network administrator must manually restore the ACL after the class has ended.

3.4 Describing policies for issuing capabilities

In our method, capabilities are issued according to a policy that is described by network administrators. The policy is described as a list of rules. Each rule consists of the permission to issue capabilities, user name, domain name, and service name in the following syntax.

```
permission   user   domain   service
```

The field `permission` is either `allow` or `deny`. Fields `user` and `domain` can include wildcards such as "`*`". The policy is evaluated from top to bottom. If the field `permission` of the matching rule is `allow`, we issue a capability. If the field `permission` is `deny` or no rules are matched, we do not issue a capability.

3.5 Format of a capability

Fig. 3 shows the format of a capability in our method. The capability consists of a version number, an external server identifier, a time to live (TTL), and an authentication code. The version number of this format is 1. The server identifier consists of the IP address of a server host that the server works on and the number of the port where the server is expecting incoming request of connections. The TTL is the time since the epoch (00:00:00 UTC, January 1, 1970) measured in seconds. The authentication code is a digital signature for preventing the forgery of capabilities. The authentication code requires a secret key shared by a DNS cache server and a router. The authentication code is a 16-octet number that is computed from the version number, IP address,

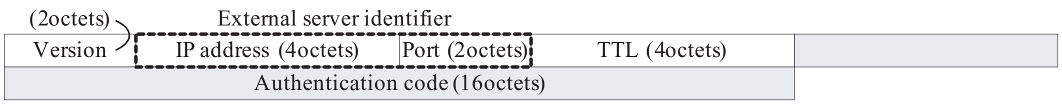


Fig. 3. Format of a capability.

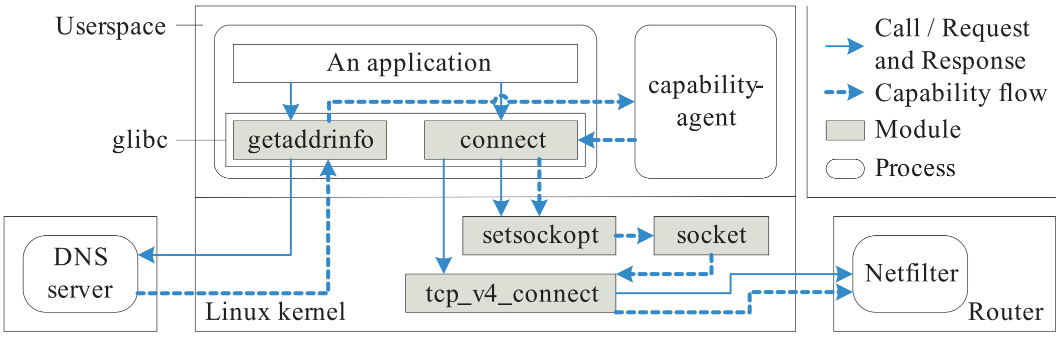


Fig. 4. Capability flows

port number, and TTL plus the secret key. As a digital signature algorithm, we use HMAC-MD5.

To carry capabilities, we have extended the DNS message format and defined a new IP option[1].

3.6 Capability flows

We have implemented capability-based egress NAC in hosts that used the Linux operating system. Fig. 4 shows capability flows.

When a process tries to resolve a server's domain name, firstly, the resolver function (`getaddrinfo`) is called. Secondly, it creates a DNS query message with a domain name, a user name, and a service name. Thirdly, the resolver signs the message with TSIG[12] to prevent forgery. TSIG is a protocol that signs DNS messages based on shared secret keys and one-way hash functions[2]. Finally, the resolver sends the DNS query message to the DNS server.

When the DNS server receives the query message, it verifies that the message is correct with TSIG and extracts the domain name, the user name, and the service name. After that, the DNS server evaluates the policy with the domain name, the user name, and the service name. If the policy allows the access, the DNS server issues capabilities and generates a DNS answer message with the capabilities. To handle TSIG and issue capabilities, we have modified the DNS server (name daemon, named in short) of BIND9[13]. The size of modified part is about 1500 lines.

[2] HMAC-MD5 is used by default.

When the resolver receives a DNS answer message from the DNS server, firstly, it verifies that the message is correct with TSIG. Secondly, it extracts capabilities from the DNS message and passes the capabilities to capability-agent through a UNIX domain socket. Finally, it returns IP addresses to the caller.

To notify capabilities to the Linux kernel, we use wrappers for the systemcall `connect`, and `sendto`. These wrappers get the capability from capability-agent for an appropriate destination and issues the systemcall `setsockopt` to save the capability into the socket before calling the original systemcall `connect` or `sendto`.

To send capabilities to a router, we modified the function `tcp_v4_connect` in the TCP layer and the function `udp_sendmsg` in the UDP layer in the Linux kernel. When the functions `tcp_v4_connect` or `udp_sendmsg` is called, it checks the socket. When the socket has a capability, the function attaches the capability to the IP option field of the outgoing IP packet.

To implement our router, we use netfilter[14], a packet filter in Linux. To check capabilities in the IP option, we have implemented a module of netfilter.

When the router receives a packet from an internal host, the module extracts a capability from the packet and verifies it. If the packet has no capability or has a wrong capability, the module drops the packet. Furthermore, the module deletes the IP option from the packet because the IP option is not needed in external networks.

4 Experiments

We have performed experiments to evaluate performance of our resolver, DNS server, Linux kernel, and router. All experiments were done on computers having an Intel Pentium4 3.0-GHz processor, 1 GB of memory, and one or two Intel PRO/1000 network interface cards. Their operating system was Debian GNU/Linux 3.0 with Linux kernel 2.6, and they were connected with a Gigabit Ethernet switch (DELL, PowerConnect 2616).

4.1 Microbenchmark for the DNS server

We ran a microbenchmark for the original named and the modified named on a native Linux kernel. This microbenchmark calls the function `getaddrinfo` 1000 times in a single process and measures the average of execution times. The results are shown in Fig. 5.

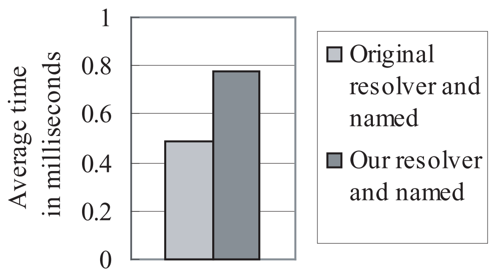


Fig. 5. Execution times of the function `getaddrinfo`.

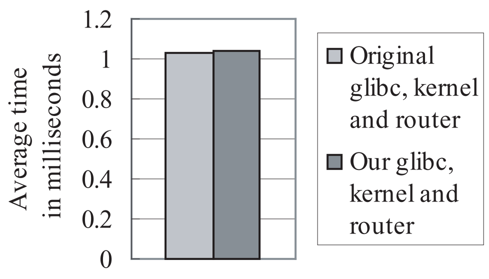


Fig. 6. Response times of the HTTP server.

The results in Fig. 5 mean that our resolver and DNS server was 0.29 millisenconds slower than the original resolver and DNS server at the time of name resolution. Most of this time was consumed by computing HMAC-MD5 in both the resolver and DNS server. This difference is much shorter than the execution time in the case of a cache miss.

4.2 Performance of the capability-enabled router

To evaluate the performance of our glibc, Linux kernel and router, we measured response times of an HTTP server (Apache 2.0) that is connected outside the router. In this experiment, the client program called the function `getaddrinfo`. After that, the client program called the systemcall `connect`, `write`, `read`, and `close` 100 times and measured the average of execution times. The HTTP server returned a 1-kilobyte file. The results are shown in Fig. 6.

The results in Fig. 6 mean that our glibc, Linux kernel and router were 0.02 milliseconds slower than the original glibc, Linux kernel and router that passed all packets. However, those differences are much shorter than typical response times of HTTP servers on the Internet, so internal users are not aware of them.

5 Conclusion

In this paper, we proposed the egress network access control based on capabilities. In our egress NAC, a user can transfer the capability to another user without any effort by the administrator.

To realize capability-based egress NAC, we used the DNS cache server and router together. The DNS server issues capabilities according to a policy. The router verifies the capabilities, and determines whether to pass or block the packets. Experimental results show that this implementation imposes a negligible overhead.

References

[1] S. Suzuki, Y. Shinjo, T. Hirotsu, K. Itano, K. Kato, Capability-based egress network access control for transferring access rights, Third International Conference on Information Technology and Applications (ICITA'2005) 2 (2005) 488–495.

[2] P. Resnick, J. Miller, PICS: Internet access controls without censorship, Communications of the ACM 39 (10) (1996) 87–93.

[3] Symantec Corporation, Symantec gateway security 5400 series refernece guide, http://www.symantec.com/.

[4] Aladdin Knowledge Systems, eSafe 4 implementation guide, http://www.eAladdin.com/.

[5] N. Zorn, Authentication gateway howto, http://www.itlab.musc.edu/˜nathan/authentication_gateway/.

[6] Y. Watanabe, K. Watanabe, H. Eto, S. Tadaki, A user authentication gateway system with simple user interface, low administrarion cost and wide applicability, IPSJ Journal 42 (12) (2001) 2802–2809.

[7] Cisco Systems Inc., Service selection gateway.

[8] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones, SOCKS protocol version 5, RFC1928.

[9] M. Leech, Username/password authentication for SOCKS V5, RFC1929.

[10] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, H. van Staveren, Amoeba – a distributed operating system for the 1990s, IEEE Computer 23 (1990) 44–53.

[11] OpenBSD, OpenSSH, http://www.openssh.com/.

[12] P. Vixie, O. Gudmundsson, D. Eastlake, B. Wellington, Secret key transaction authentication for DNS (TSIG), RFC2845.

[13] Internet System Consortium, BIND 9, http://www.isc.org/.

[14] O. Andreasson, Iptables Tutorial 1.1.19, http://www.netfilter.org/.