Journal of Information Security, 2012, 3, 86-90
http://dx.doi.org/10.4236/jis.2012.32010 Published Online April 2012 (http://www.SciRP.org/journal/jis)

Reference Encryption for Access Right Segregation and Domain Representation

Lanfranco Lopriore
Dipartimento di Ingegneria dell'Informazione: Elettronica, Informatica, Telecomunicazioni, Università di Pisa, Pisa, Italy
Email: l.lopriore@iet.unipi.it

Received January 30, 2012; revised February 29, 2012; accepted March 25, 2012

ABSTRACT

With reference to a protection model featuring processes, objects and domains, we consider the salient aspects of the protection problem, domain representation and access right segregation in memory. We propose a solution based on protected references, each consisting of the identifier of an object and the specification of a collection of access rights for this object. The protection system associates an encryption key with each object and each domain. A protected reference for a given object is always part of a domain, and is stored in memory in the ciphertext form that results from application of a double encryption using both the object key and the domain key.

Keywords: Access Right; Domain; Protection; Symmetric-Key Cryptography

1. Introduction

We shall refer to a well-known protection model featuring active entities, the *processes*, that perform access attempts to passive entities, the *objects* [1,2]. Objects are typed; the type of a given object states the set of operations that can be carried out on this object and, for each operation, the *access rights* that a process must hold to accomplish this operation successfully. At any given time, a *protecttion domain* is associated with each process: this is a collection of access rights on the objects that the process can access at that time.

A salient aspect of the protection problem is the representation of access rights and protection domains in memory. A classical solution is based on the concept of a *capability* [3,4]. This is a pair <*B*, *AR*>, where *B* is the identifier of an object and *AR* is a set of access rights for this object. A protection domain takes the form of a collection of capabilities, which correspond to the access rights included in that domain.

Capabilities are sensitive objects that cannot be treated as ordinary data [5]: we must prevent processes from modifying the access right field and add new access rights, for instance. Capabilities can be segregated into *capability segments* [6,7]. In this case, a protection domain usually takes the form of a tree, where the root of the tree is a capability segment that includes the capabilities for other capability and data segments, and the data segments are the tree leaves. Alternatively, we can take advantage of a *tag* associated with each memory cell, which specifies whether this cell contains a capability or an ordinary data item [8,9]. In a third approach, a set of passwords is associated with each object, and each password corresponds to one or more access rights. A *password capability* is a pair <*B*, *PSW*> where *B* is an object identifier and *PSW* is a password [10,11]. If a match exists between *PSW* and one of the passwords associated with object *B*, then the password capability grants its holder the access rights corresponding to that password on *B*.

In the approaches to capability segregation in memory, outlined so far, a process that holds a capability can take full advantage of this capability, independently of the capability origin. This means that segregation does not prevent a process from taking advantage of a capability obtained illegitimately by means of a fraudulent action of capability copy, for instance.

In this paper, we propose an alternative approach to access right representation in memory, which solves the segregation problem by taking advantage of a form of *symmetric-key cryptography* [12,13]. In our approach, possession of an access privilege on a given object is certified by possession of a *protected reference* (*p-reference* from now on, for short) including the specification of a collection of access rights for this object. P-references are never stored in memory in plaintext. Instead, the protection system associates an encryption key, called the *object key*, with each object, and a further encryption key, the *domain key*, with each domain. A p-reference for a given object is always part of a protection domain and is stored in memory in the ciphertext form that results from application of a double encryption using both the object key and the domain key.

2. The Protection System

2.1. Protected References

Let T be an object type, let S_0, S_1, $\cdots$ be the operations that can be executed on an object of type T, and let AR_0, AR_1, $\cdots$ be the access rights defined by T. For each given operation S_m, the definition of type T states the subset of access rights AR_0, AR_1, $\cdots$ that is necessary to accomplish that operation successfully. P-reference R takes the form $R = <B, AR>$, where AR is a bit configuration that specifies a collection of access rights for object B: if the i-th bit of AR is asserted, R grants access right AR_i on B.

From now on, we shall use an underline to denote a ciphertext. Let k_B be the encryption key associated with object B, and k_D be the encryption key associated with the domain D of p-reference $R = <B, AR>$. **Figure 1** shows the transformation of R into ciphertext quantity $\underline{R}$. The transformation proceeds as follows. Let $\underline{B}$ be the result of encrypting quantity $\underline{B}$ by using a symmetric-key cipher with key k_D, and let $\underline{AR}$ be the result of encrypting pair $<\underline{B}, AR>$ by using a symmetric-key cipher with key k_B. Quantity $\underline{R}$ is given by relation $\underline{R} = <\underline{B}, \underline{AR}>$.

Figure 2 shows the reverse transformation of ciphertext quantity $\underline{R} = <\underline{B}, \underline{AR}>$ into the corresponding plaintext p-reference R. The transformation proceeds as follows. Domain encryption key k_D is used to decrypt quantity $\underline{B}$ into object name B. Then, the object key k_B associated with object B is used to decrypt quantity $\underline{AR}$. Let $<\underline{B}^*, AR>$ be the result of the decryption. Quantity $\underline{B}^*$ is compared with $\underline{B}$ to validate AR; if a match is found, validation is successful and p-reference R is given by pair $<B, AR>$.

2.2. Processes, Domains and Objects

When a new process is generated that has no parent (*i.e.* a process directly generated by the kernel), a new domain is created and is associated with this process. When a process generates a child process, the new process is assigned the same domain as the parent process. Thus, the tree structure originated by recursive actions of child process generation is entirely confined within the boundaries of the same domain, which is the domain of the root process. All the processes in the tree are *tightly coupled*, *i.e.* they share the same domain and consequently, the same domain key.

When a process creates a new object, it receives a p-reference for this object with full access rights. If the identifier of every given object is equal to the address of this object in the virtual space, a simple approach to generation of new object identifiers is a sequential generation, according to which objects are allocated at increasing virtual addresses, and the address of a given object is equal to the address of the previous object incremented by the length of the previous object.

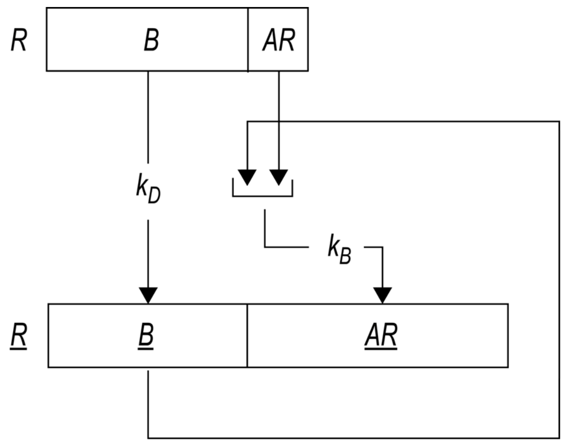

Figure 1. Transformation of plaintext p-reference $R = <B, AR>$ into ciphertext quantity $\underline{R} = <\underline{B}, \underline{AR}>$.

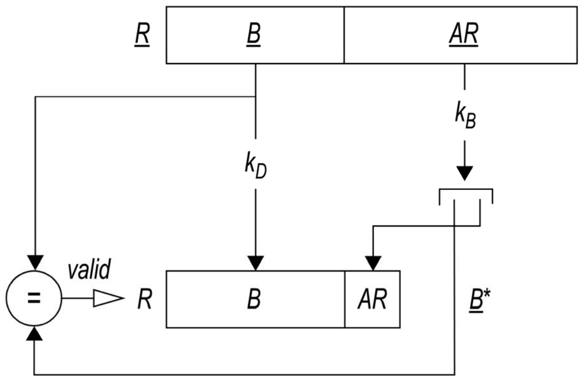

Figure 2. Transformation of ciphertext quantity $\underline{R} = <\underline{B}, \underline{AR}>$ into the corresponding plaintext p-reference $R = <B, AR>$, and validation of the result.

2.3. Accessing Objects

Let P be a process, let D be its domain, and let $\underline{R}$ be a p-reference stored in ciphertext form in the memory area reserved for P. In order to take advantage of $\underline{R}$, process P must preventively translate it into the corresponding plaintext $R = <B, AR>$ so that both the name B of the object referenced by $\underline{R}$ and the access rights AR granted by $\underline{R}$ on this object become visible. Of course, after translation into plaintext, p-reference R is a sensitive information item that must be stored in a protected memory region. To this aim, the protection system reserves a *protection table* for each given process; each entry of the protection table can contain a p-reference in plaintext.

Process P can load a p-reference into an entry of its own protection table by executing the *LoadRef*(*addr*, *i*) protection system primitive. Execution of this primitive causes the actions necessary to translate ciphertext p-reference $\underline{R}$ stored in memory at address *addr* into a plaintext by using the key k_D of the domain D of P (see **Figure 2**); the result $R = <B, AR>$ of the translation is stored into the i-th protection table entry.

Let T be the type of object B and S_0, S_1, $\cdots$ be the operations defined by this type. The operation call primitive *Call*(m, i) starts up execution of operation S_m on B. Argument i is the index of the protection table entry containing a p-reference for B. The value AR of the access right field of this p-reference is transmitted to S_m as an input parameter. The actions involved in the execution of S_m will include the access right checks necessary to ascertain whether AR contains the access rights required to accomplish S_m successfully. If this is not the case, execution of S_m fails and generates an exception of violated protection.

2.4. Transferring P-References

Let P_1 and P_2 be tightly coupled processes, and let D be their common domain. Suppose that P_1 holds p-reference $\underline{R}$. In order to grant P_2 the access permissions included in $\underline{R}$, P_1 transfers $\underline{R}$ to P_2 by a simple action of a memory copy. In fact, the two processes share the same domain key k_D. By issuing the *LoadRef*() primitive, P_2 will decrypt $\underline{R}$ into the corresponding plaintext p-reference R. Then, by issuing the *Call*() primitive, P_2 will be in the position to use R for object access.

Let us now suppose that P_1 and P_2 belong to different domains, D_1 and D_2, and let k_1 and k_2 be the keys of these domains. If P_1 transfers a copy of $\underline{R}$ to P_2, no access privilege is actually granted to P_2. In fact, if P_2 issues *LoadRef*() to decrypt $\underline{R}$, the $\underline{B}$ component of $\underline{R}$ will be decrypted by using the key k_2 associated with domain D_2 instead of the key k_1 that was originally used to encrypt $\underline{B}$. The result of this translation will be the identifier of an arbitrary object whose key will be used to translate the $\underline{AR}$ component of $\underline{R}$ into plaintext. Of course, validation of the result of this translation is destined to fail.

Instead, the copy of a p-reference between two processes of different domains must be preceded by a conversion of the p-reference, from the domain of the granting process to the domain of the process that receives the p-reference (the *target domain*). To this aim, an object called the *encryption channel* is associated with each domain. The protection system maintains the association between each encryption channel and the key of the corresponding domain. P-reference conversion will be actually obtained by taking advantage of the *StoreRef*(i, j, *addr*) primitive. Arguments i and j of this primitive are the indexes of two entries in the protection table of the process issuing the primitive; these entries contain a p-reference for the encryption channel of the target domain and the p-reference to be converted. Execution of this primitive converts this p-reference into a ciphertext using the key associated with the encryption channel, and stores the result of the conversion into memory at address *addr*.

In our previous example, let $\underline{EC}$ be the ciphertext form of a p-reference for the encryption channel of the target domain D_2. In order to grant the access privileges in $\underline{R}$ to P_2, process P_1 will issue the *LoadRef*() primitive twice, to translate $\underline{EC}$ and $\underline{R}$ into plaintext and load the results EC and R of these translations into free protection table entries. Then, P_1 will issue the *StoreRef*() primitive to convert R into a ciphertext using the domain key k_2 associated with EC. Finally, P_1 will copy this ciphertext to process P_2, which will be able to take advantage of it, as it is now part of its own domain, D_2.

3. Discussion

P-references are stored in memory as ordinary information items, albeit in ciphertext form. Consequently, a process may well try to modify an existing p-reference and amplify the access rights it contains, or attempt to generate a p-reference for an existing object from scratch. A process may even perform fraudulent actions of p-reference copy. Storage of p-references in the stack and heap memory areas results in occasions for application of well-known techniques for data stealing [14,15], for instance. As a matter of fact, these attempts to p-reference manipulation are destined to fail.

3.1. Forging P-References

Let us refer to p-reference $R = <B, AR>$, and let $\underline{R} = <\underline{B}, \underline{AR}>$ be the corresponding ciphertext in memory. The $\underline{AR}$ field of $\underline{R}$ is the result of application of an encryption involving both quantity $\underline{B}$ and the access right specification AR (see **Figure 1**). We shall hypothesize that the cipher guarantees a careful mixing of the bits of $\underline{B}$ and AR. In a situation of this type, it is impossible to modify the resulting ciphertext $\underline{AR}$ for the sole portion corresponding to the access rights without corrupting quantity $\underline{B}$. In order to use the modified $\underline{R}$ to access B, the process holding $\underline{R}$ must preventively issue the *LoadRef*() primitive to translate $\underline{R}$ into plaintext and load the result R into the protection table. The actions involved in the translation from $\underline{R}$ to R include a validation of AR that involves quantity $\underline{B}$ (see **Figure 2**). If $\underline{AR}$ has been modified, the probability of a casual match leading to successful validation depends on the size of object identifiers; for large identifiers, e.g. 64 bits, this probability is vanishingly low [16]. When *LoadRef*() fails, an exception of violated protection is generated.

Of course, similar considerations can be made for a process attempting to forge a new p-reference for an existing object. The process will have to issue *LoadRef*() to translate the forged p-reference into plaintext and load it into the protection table; in this case, too, validation of the result of this translation is destined to fail.

3.2. Stealing P-References

Let D_1 be the domain of process P_1, and let k_1 be the key associated with this domain. Suppose that P_1 holds a p-reference in the ciphertext form $\underline{R}$ = <$\underline{B}$, $\underline{AR}$> obtained by using key k_1. Suppose also that a different process P_2, which is part of domain D_2, steals a copy of $\underline{R}$ from P_1. In order to take advantage of $\underline{R}$ and access the object it references, P_2 has to issue the *LoadRef*() primitive to translate $\underline{R}$ into plaintext and load the result of this translation into its own protection table. In the execution of *LoadRef*(), quantity $\underline{B}$ will be decrypted using the key k_2 associated with domain D_2 instead of the key k_1 that was originally used to encrypt $\underline{B}$. The result of this action will be an object identifier *BB*. In the hypothesis of large object names, the probability that *BB* be the identifier of an existing object will be low, and the search for the decryption key k_{BB} associated with *BB* is likely to fail. Even in the improbable situation that *BB* is a valid identifier, the corresponding key k_{BB} will not match the key that was used to generate $\underline{AR}$, and consequently, validation of the result of the translation of $\underline{AR}$ into plaintext is destined to fail. In both cases, *LoadRef*() terminates with an exception of violated protection.

4. Concluding Remarks

With reference to a protection environment featuring processes, objects and domains, we have approached the salient aspects of the protection problem, domain representation and access right segregation in memory. We have proposed a solution based on protected references, each consisting of the identifier of an object and the specification of a collection of access rights for this object. The protection system associates an encryption key with each object and each domain. A p-reference for a given object is always part of a domain, and is stored in memory in the ciphertext form that results from application of a double encryption using both the domain key and the object key.

Double encryption enhances security only marginally [17]; in our protection system, we take advantage of a double encryption and the duality between object keys and domain keys to guarantees the practical impossibility to acquire access permissions for an existing object by forging a new p-reference for this object or modifying an existing p-reference to amplify the access rights it contains. Furthermore, a process running in a given domain cannot take advantage of a p-reference encrypted as part of a different domain. In sharp contrast with capability and password-capability systems, this aspect of p-reference segregation in memory prevents the stealing of access permissions between processes of different domains. On the other hand, two processes that share the same father process belong to the same domain; these processes are considered mutually trustworthy, and the transfer of a p-reference between them can be obtained by a simple action of memory copy, at low processing time cost.

REFERENCES

[1] L. Lopriore, "Access Control Mechanisms in a Distributed, Persistent Memory System," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 13, No. 10, 2002, pp. 1066-1083. doi:10.1109/TPDS.2002.1041883

[2] R. S. Sandhu and P. Samarati, "Access Control: Principle and Practice," *IEEE Communications Magazine*, Vol. 32, No. 9, 1994, pp. 40-48. doi:10.1109/35.312842

[3] H. M. Levy, "Capability-Based Computer Systems," Butterworth-Heinemann, Oxford, 1984.

[4] I. Kuz, G. Klein, C. Lewis and A. Walker, "CapDL: A Language for Describing Capability-Based Systems," *Proceedings of the* 1*st ACM Asia-Pacific Workshop on Systems*, New Delhi, 30 August-3 September August 2010, pp. 31-36. doi:10.1145/1851276.1851284

[5] M. de Vivo, G. O. de Vivo and L. Gonzalez, "A Brief Essay on Capabilities," *SIGPLAN Notices*, Vol. 30, No. 7, 1995, pp. 29-36. doi:10.1145/208639.208641

[6] G. Klein, *et al*., "seL4: Formal Verification of an OS Kernel," *Proceedings of the* 22*nd ACM Symposium on Operating Systems Principles*, Big Sky, 11-14 October 2009, pp. 207-220. doi:10.1145/1629575.1629596

[7] E. I. Organick, "A Programmer's View of the Intel 432 System," McGraw-Hill, New York, 1983.

[8] P. G. Neumann and R. J. Feiertag, "PSOS Revisited," *Proceedings of the* 19*th Annual Computer Security Applications Conference*, Las Vegas, 8-12 December 2003, pp. 208-216. doi:10.1109/CSAC.2003.1254326

[9] L. Lopriore, "Capability Based Tagged Architectures," *IEEE Transactions on Computers*, Vol. C-33, No. 9, 1984, pp. 786-803. doi:10.1109/TC.1984.1676495

[10] M. D. Castro, R. D. Pose and C. Kopp, "Password-Capabilities and the Walnut Kernel," *The Computer Journal*, Vol. 51, No. 5, 2008, pp. 595-607. doi:10.1093/comjnl/bxm124

[11] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell and J. Liedtke, "The Mungi Single-Address-Space Operating System," *Software*: *Practice and Experience*, Vol. 28, No. 9, 1998, pp. 901-928. doi:10.1002/(SICI)1097-024X(19980725)28:9<901::AID-SPE181>3.0.CO;2-7

[12] M. Stamp, "Information Security: Principles and Practice," 2nd Edition, Wiley, Hoboken, 2011. doi:10.1002/9781118027974

[13] J. Burke, J. McDonald and T. Austin, "Architectural Support for Fast Symmetric-Key Cryptography," *Proceedings of the* 9*th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, 12-15 November 2000, pp. 178-189. doi:10.1145/378993.379238

[14] N. Tuck, B. Calder and G. Varghese, "Hardware and Binary Modification Support for Code Pointer Protection

from Buffer Overflow," *Proceedings of the* 37*th Annual IEEE/ACM International Symposium on Microarchitecture*, Portland, 4-8 December 2004, pp. 209-220. doi:10.1109/MICRO.2004.20

[15] Y. Younan, F. Piessens and W. Joosen, "Protecting Global and Static Variables from Buffer Overflow Attacks," *Proceedings of the* 4*th International Conference on Availability, Reliability and Security*, Fukuoka, 16-19 March 2009, pp. 798-803. doi:10.1109/ARES.2009.126

[16] M. Anderson, R. D. Pose and C. S. Wallace, "A Password-Capability System," *The Computer Journal*, Vol. 29, No. 1, 1986, pp. 1-8. doi:10.1093/comjnl/29.1.1

[17] P. Gaži and U. Maurer, "Cascade Encryption Revisited," *Proceedings of the* 15*th International Conference on the Theory and Application of Cryptology and Information Security*, Tokyo, 6-10 December 2009, pp. 37-51. doi:10.1007/978-3-642-10366-7_3