Protecting and Confining Information with Speedos

James Leslie Keedy
Eislebener Str. 20, 28329 Germany
email: keedy@jlkeedy.net
formerly Professor/Honorary Professor at the University of Newcastle, NSW; Monash University, Melbourne, the Technical University of Darmstadt, the University of Bremen and the University of Ulm

Abstract

This paper introduces some of the key mechanisms which the Speedos Operating System provides to ensure that users can secure and protect their information and programs. After a short introduction which stresses the importance of security in the modern computer world and the costs which can face those who are forced to use conventional computers, it outlines the key mechanisms supported in the Speedos system. These include capabilities based on the rigorous use of the information-hiding technique in a persistent virtual memory, borrowing from and updating ideas which were originally introduced in the Monads Systems. But above all the paper introduces and explains the idea of "qualifiers", a key new technique for solving protection problems. Examples are provided to illustrate some of the power of qualifiers.

Keywords

Architectural security, capability systems, monitoring software, decoys, capability revocation, confining information, ransomware.

1 Introduction

One of the most serious weaknesses of conventional operating systems is their inability to confine the information stored in them in such a way that it cannot be stolen or modified as a result of cyber-attacks such as

- spyware attacks;
- attacks aimed at destroying critical infrastructure by manipulating key controls;
- attacks on hospitals and similar institutions;
- attacks on banks and similar institutions;
- attacks on power stations;
- attacks on rail and other transport systems;
- theft of credit card information and similar;
- any other situation which requires that information can be securely confined, with access available only for authorised users.

On 16th August 2023 the German Federal Criminal Office issued a press information sheet which indicated that in 2022 the German police registered 136,865 cases of cybercrime, which was an increase of 6.5 % over the previous year, with indications that there would be a further increase of around 8 % in 2023. Calculations of the organisation BITKOM, a German employer organisation specialising in Cybercrime, based on questionnaires to their member companies, indicated that damages of about 203 Billion Euros in 2022 were incurred in Germany, which was around double so high as in 2019. Some of these damages arose from ransomware attacks, and although Speedos was not specifically designed to combat these, I am relatively confident that these too can be prevented by Speedos; but until Speedos systems exist in reality, it is impossible to be 100 % sure. Leaving ransomware aside, there is an enormous range of attacks where Speedos can prevent potential damage.

The background of the Speedos Operating System is explained on the Speedos website (see https://www.speedos-security.org/introduction-to-speedos.html). One of the main aims of the Speedos operating system is to provide weapons which enable programmers and users to protect their information. The two most important weapons are capabilities and qualifiers. First we briefly describe these two mechanisms; then we provide some examples.

2 Capabilities

In the pre-RISC era the most promising security strategies at the architectural level were undoubtedly found in the early capability systems, e.g. [1, 2, 3, 4]. A capability, first proposed by Dennis and van Horn [5], is a data structure which locates an object and describes what access rights the presenting process can exercise over the object (e.g. for memory segments: read and/or write and/or execute rights together with a length field which can be used to check the bounds of the object, thus preventing accesses outside the segment). Most early capability system designers chose variable length segments of memory as the objects to be protected; these allow different processes to have separate capabilities for the same segment, which could, if appropriate, have different access rights[1]. Individual segments might be small or large. Hence a capability system which supports both large and small segments [6] is ideal, especially since capabilities themselves are normally relatively small objects.

Capabilities exist at two levels in Speedos. *Segment capabilities* protect segments (with access rights read, write and/or execute and a length check). These are used internally within modules. *Module capabilities* protect information-hiding modules [7, 8, 9]) that can have multiple entrypoints. Figure 1 illustrates how a bank accounts file might be designed as an information-hiding module.

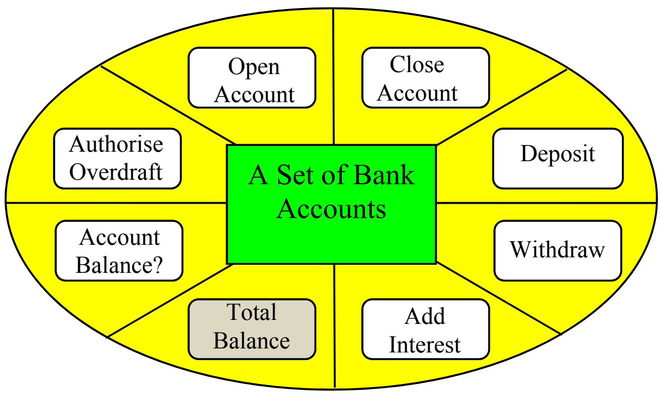

Figure 1 A Bank Accounts File Module

Speedos does not have a conventional file system. Instead it has a *persistent virtual memory* which is populated entirely by information-hiding modules and by persistent processes and their threads [10]. When a persistent process is active its threads make in-process *inter-module calls* to modules. To make such calls the thread must present an appropriate module capability. Such a capability consists primarily of a unique module identifier and a bitlist of access rights, together with a valid bit (see Figure 2).

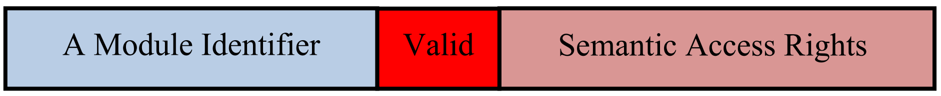

Figure 2: A Module Capability

[1] This contrasts with conventional systems, which place the access rights in page tables and thus determine that all users share the same access rights.

The access rights in module capabilities are referred to as *sematic access rights.* Semantic access rights allow software designers to assign roles to actors, leading to better protection of the information in file modules. For example access to information in the bank accounts module (Figure 1), might be as shown in Figure 3.

	Teller	Branch Manager	H.O. Accountant	H.O. Auditor
Open Account	√	√	x	x
Close Account	√	√	x	x
Deposit	√	√	x	x
Withdraw	√	√	x	x
Transfer	√	√	√	x
Add Interest	x	x	√	x
Authorise Overdraft	x	√	x	x
Customer Number	√	√	x	√
Overdraft Limit	√	√	√	√
Current Balance	√	√	√	√

A tick indicates that the subject at the head of the column may carry out the operation in the corresponding row.

Figure 3: Access Rights expressed as Semantic Operations

The inter-module call is carried out directly by the Speedos system kernel. The kernel checks that the module exists, that the capability is valid and that the appropriate permission bit in the access rights bitlist is set. If the call is permitted the kernel then transfers thread execution to the nominated entrypoint of the module. If the called module is located on a different node (which the kernel can check from the module identifier) the call is made as a *remote* inter-module call. How this actually occurs is described in volume 2 chapter 28 section 3 of the book "Making Computers Secure"[2].

How capabilities themselves are protected in Speedos is described in [11].

3 Qualifiers

Semantic access rights in capabilities provide a more powerful mechanism than is found in conventional systems: users can positively define who can access their information and programs in a more refined way. But capabilities cannot guarantee that information is safe from attacks by cyber-criminals! For this and other purposes Speedos provides a quite different mechanism, known as "qualifiers". The idea for this initially came from Keedy's programming language work, in a paper which appeared in 1997 under the name "attribute types" [12] and is now known as "qualifying types", or simply qualifiers.

3.1 Call-in Brackets

A qualifier is an instance of a type which, in addition to its normal methods, has "bracket" methods that can qualify the behaviour of a different object (the target object) with which it is associated. These are automatically activated methods (called "call-in" methods) which

[2] Available for download from the Speedos website https://www.speedos-security.org/

"catch" a call to a method of the target instance. To illustrate this, we first consider a normal call from a client object to a target object (Figure 4). At the Speedos level the objects concerned are all information-hiding modules (e.g. information hiding files)[3].

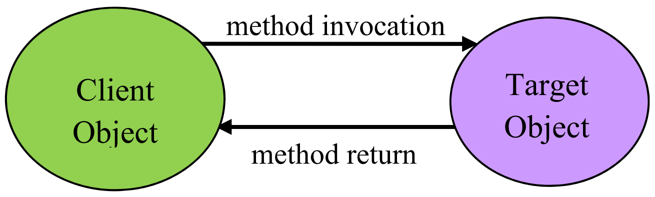

Figure 4: A Normal Method Invocation

Now we add a call-in bracket method (which has been associated with the target object by its owner), see Figure 5.

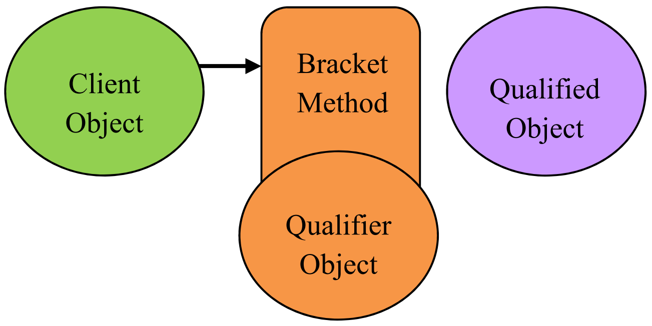

Figure 5: A Qualifying Type with a Call-In Bracket Method

The bracket method can access the parameters which the client object intended to pass to the qualified object. But it has no access to the data of either the client object or of the qualified object.

Additional code can be added before calling the qualified object (in the part of the bracket method called a *prelude*), see Figure 6. This code might for example access synchronising variables in the data of the qualifier, thus causing an unsynchronised qualified object to be synchronised. Or from the security viewpoint it might for example maintain a log of calls to the qualified object which can later be printed out or analysed by another computer program to detect attempts to hack the qualified object.

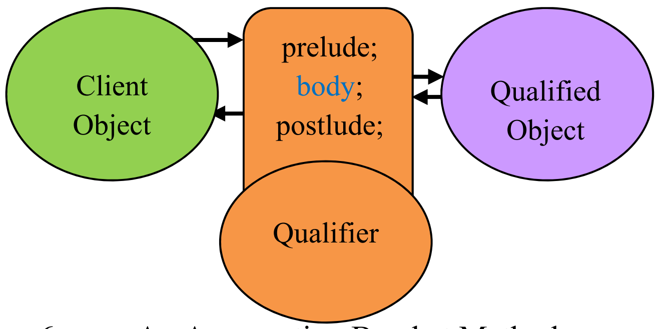

Figure 6: An Augmenting Bracket Method

The code of a call-in bracket method can include a special **body** call, which indicates the point in the code (if any) at which the bracket method allows the call to proceed to the

[3] Timor supports qualifiers at both the internal programming level and at the Speedos information-hiding module level. In the latter case the Speedos kernel provides the necessary support for qualifiers.

target method. This can appear at any point in the code of the bracket method, but may only once be called dynamically. When the method of the qualified object has completed its task, it returns to the postlude section of the call-in method (i.e. the statements following the body call). In the postlude section it can, for example, reset the synchronisation variables. This option, which augments the code of the qualified object, is shown in Figure 6.

The body call can optionally be placed in a conditional statement (see Figure 7) in which case the call-in bracket can be used to carry out tests (e.g. security checks) to determine whether the qualified object may be called or not.

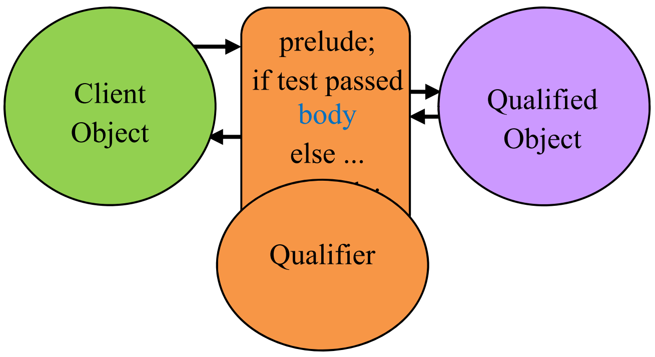

Figure 7: A Testing Bracket Method

The call-in method can optionally let the call through, but it can also change or invalidate its parameters. A further possibility is that the call-in bracket can simply return to the caller, as is shown in Figure 8. In this case it can serve as a decoy, which can be used as a disinformation technique.

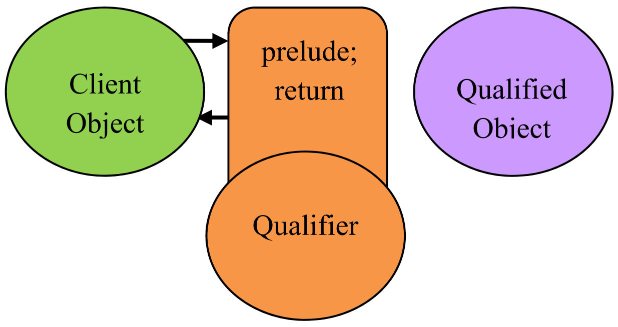

Figure 8: A Replacing Bracket Method

More than one qualifier can be associated with a qualified object. In this case there is a defined order such that the first is invoked as a result of a routine call from a client object, the next is then invoked if this makes a body call, etc.; a body call from the final qualifying object (if it ever happens) results in the target object being called. The postludes are executed in reverse order.

3.2 Call-out Brackets

A qualifier can also have "call-out" methods [13]. The principle of call-out bracket methods is similar to that of call-in methods, except that

a) they are triggered by a call *from* a qualified object to some other object (the call-out object);

b) a **call** statement (cf. the **body** statement for call-in methods) is used if the call-out bracket decides to pass the call on to the call-out object.

The basic concept is illustrated in Figure 9, where a qualifying object has both call-in and call-out bracket methods. However, a qualifier can be programmed to have only call-in or only call-out routines if that is appropriate.

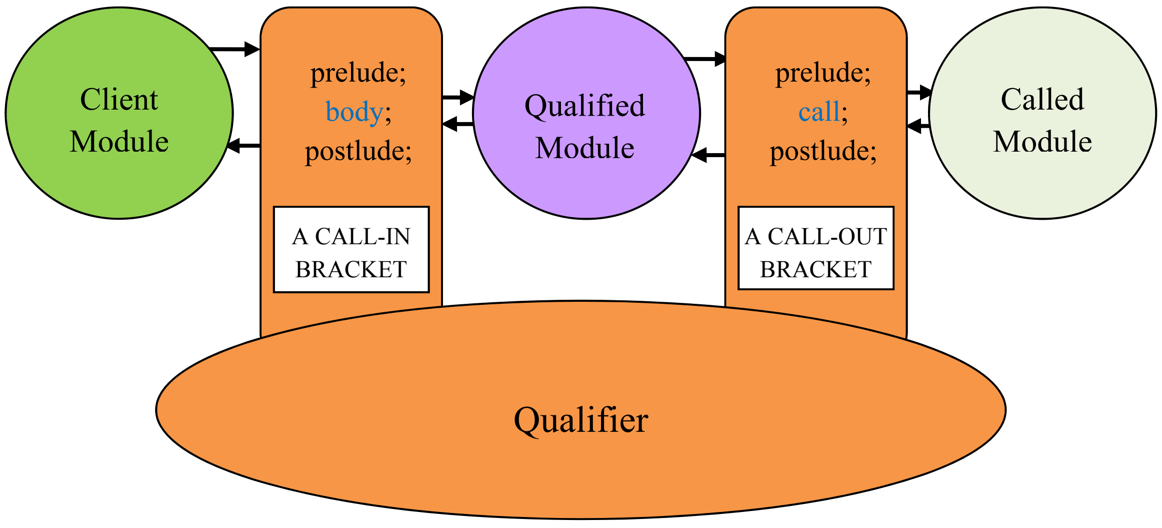

Figure 9: A Qualifier with Call-In and Call-Out Bracket Methods

Call-out brackets can be freely programmed to include or omit a **call** statement, and can optionally place it in a conditional statement.

Call-out brackets can play a significant role towards solving the *confinement problem*. On an individual module basis they can dynamically examine what information is being passed out of a module. They can, for example, invalidate or destroy capabilities, and they can even replace them with "false" capabilities, as a disinformation technique.

It is important that Speedos qualifiers can be *dynamically* associated with their target modules at run-time and that these associations can be dynamically changed at any time by those with permission to do so.

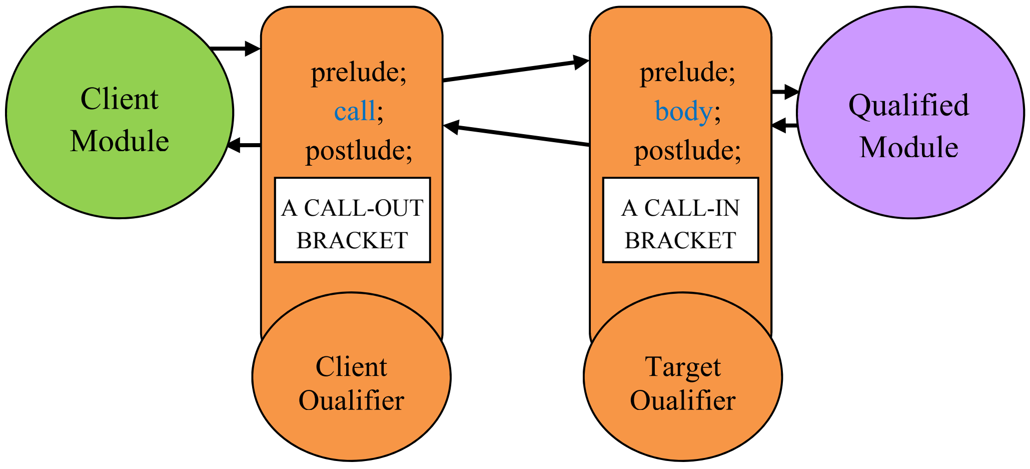

Figure 10: A Client with Call-Out and a Target with Call-In Brackets

3.3 Combining Call-in and Call-out Brackets

At first sight it might be thought that call-out routines are superfluous, with the argument that they could be implemented as call-in brackets of the call-out object. However, this is not the case, because a call-in bracket is activated whenever the qualified object is called, while a call-out bracket is activated each time the qualified object makes a call to another object, not each time the called object is invoked. However both a client object and its qualified object can be qualified (usually, but not necessarily, by different qualifier objects), as is shown in Figure 10.

At the local object level qualifiers are managed by the compiler, but at the module level they are managed by the operating system, i.e. by the Speedos kernel. When associated with files the call-out methods are particularly significant, since these can be used to prevent information from being passed to unauthorised recipients.

Qualifiers, which can also qualify other qualifiers, give Speedos very powerful tools for protecting information. In particular they provide a fine-grained solution for the access control problem [14] (via semantic files) and a new and precise solution for the confinement problem. The latter is usually considered as a firewall problem. However, qualifiers with call-out bracket methods provide a much more effective solution which can in effect provide flexible individual user-level firewalls for individual modules.

4 Aids for checking and preventing Security Breaches

In order to carry out security checks, software (e.g. bracket routines, but also normal modules) often needs information about the environment in which it is working. For example when checking whether a capability has been revoked for a user, it is necessary to know who the user is, and whether he owns the thread currently trying to use the capability. To satisfy this and many similar requirements, there are kernel instructions which provide such information.

4.1 Checking Application Modules

In his doctor thesis[4] my former Ph.D. student Klaus Espenlaub [15] listed three groups of kernel instructions for this purpose, to which I later added a fourth group.

i) The first group of kernel calls returns information directly related to the current environment of a thread, in the form of world-wide unique module or thread identifiers[5]. These kernel calls can, for example, return the unique identifier of the currently active thread, the identifier of the current file module, of the current code module, of the file module from which it was called, of the calling code module, in the case of a callout module the target file module being called and of the code module being called.

ii) The second group of kernel instructions identifies the *owner* of each of the above, in the form of a unique container identifier)[6].

   The kernel obtains this information from the red tape at the beginning of the corresponding container. For example the kernel call `current_file_owner` is the owner of the container in which the currently active data file is located.

iii) The third instruction group returns the number of the semantic routine (entry point) of the currently active module, of the semantic routine which called this, and of the semantic routine currently being invoked.

iv) A fourth group of environmental instructions, known as the *calling rights*, is needed in order that bracket routines can more thoroughly check the access rights associated with the *target call* than was envisaged by Espenlaub. These include semantic rights, metarights and capability rights, which are obtained by the kernel from the calling capability, the remaining rights from the Thread Security Register.

[4] downloadable at http://vts.uni-ulm.de/doc.asp?id=5333

[5] A module/thread identifier is its full 192 bit identifier, i.e. Node #, Disc # and Container # (including index). It is not a capability.

[6] The index field, which normally signifies the module number within a container, is set to -1 when an entire container is being identified. The number of the first container for a new user identifies the user uniquely throughout his existence in the system and has the index value -2.

5 The Thread Security Register[7]

The Thread Security Register (TSR) is a pseudo register maintained by the kernel as part of the state of each user thread. It holds a set of rights currently associated with the thread. Its current values are stored at the base of the kernel's thread stack and are recorded in the linkage segment on each inter-module call and on bracket routine activations (and restored on the corresponding returns). Its current values are available only indirectly to active modules and bracket routines via kernel instructions. Its content is extremely security sensitive and it is fully protected from direct user access.

The permissions in the TSR follow the same rules as those for access rights in capabilities. Initially all the rights are set (implemented as 1 in the TSR), i.e. all permissions can initially be used, but can be reduced (unset/turned off, i.e. with the value 0). A permission which has been turned off cannot be explicitly turned back on. To reduce the rights, the kernel uses an intersection operation.

The rights fall into four groups (thread control rights, confinement rights, environmental rights and capability accessibility rights). These are explained in Chapter 25 of volume 2 of [16].

When a thread is initially created, all the rights are set. As the thread proceeds, some or even all of these rights may be removed or changed. An application cannot restore its own rights. The removal of rights can be effected by an application or bracket routine using a kernel refinement instruction. Applications and bracket routines can also examine the current contents by executing kernel instructions.

Here are some examples of the rights:

permit_remote_node: When unset, the kernel prevents a thread from being transferred to another node.

permit_foreign_calls: When unset, the kernel prevents calls to modules owned by users other than the owner of the thread.

permit_foreign_file_caps: When unset, the kernel prevents the thread from making use of any file capabilities for modules owned by users other than the owner of the thread.

permit_foreign_code_caps: When unset, the kernel prevents any use of code capabilities for modules owned by users other than the owner of the thread;

permit_foreign_thread_caps: When unset, the kernel prevents the thread from making use of any thread capabilities for modules owned by users other than the owner of the thread.

permit_download: When unset, this right prevents the thread from initiating downloads.

permit_upload: When unset, the Container Manager prevents the thread from initiating uploads.

permit_subthreads: When unset, the thread cannot create subthreads.

permit_callbacks: When unset, the thread cannot invoke or support call-back routines.

permit_websites: When unset, the thread cannot access non-Speedos websites.

permit_mail: When unset, the thread cannot access non-Speedos email systems.

permit_FTP: When unset, the thread cannot access non-Speedos FTP facilities.

permit_other_internet: When unset, the thread cannot access any non-Speedos Internet facilities.

---

[7] The following description of the Thread Security Register is considerably oversimplified. For more details see chapter 26 section 4 of Making Computers Secure volume 2 which can be downloaded from the Speedos website https://www.speedos-security.org/.

6 Some Examples

6.1 Monitoring and Logging Access to Important files

Maintaining log files which hold information about security-related events can be an important step towards discovering what might go wrong in a system. For example it might be useful to discover who or what has accessed (or attempted to access) an important file. To do this, a call-in bracket can be useful. Such a routine can be placed before the file to be monitored, as is shown in Figure 11.

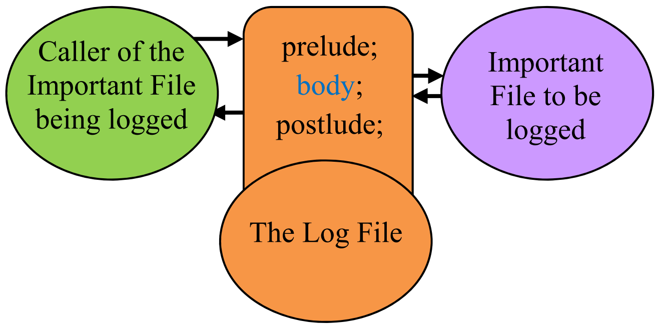
Caller of the Important File being logged
prelude; body; postlude;
Important File to be logged
The Log File

Figure 11: Logging and Monitoring a File

Unbeknown to the caller of the Important File, the call-in bracket intercepts the call and can make an entry in its Log File. This can occur in the code of the prelude. If the user's call need not be prevented the bracket routine can then execute the *body* call, allowing the call to proceed to the Important File. The content of the logged entry is determined by the format defined in the specification of the Log File. This can for example be the unique identifier of the caller, the time of the call, the node from which the call was initiated, etc. Such information can be gathered in the way described in section 4.

When the method of the Important File has completed its task, it returns to the postlude. Since "Important File" is a normal file it can have normal (non-bracket) methods which can be used later to read and for example to analyse who is accessing the file and perhaps what he hopes to achieve.

6.2 Using a decoy

If the owner of Important File wishes to prevent the caller from accessing the Important File, the simplest way to achieve this simply *not* to issue a *body* call, as is shown in Figure 8. If he wishes to give the impression that the user's call was successful, he might return information which gives the impression that the caller actually reached the Important File. This might, but need not, provide a decoy, which realistically returns false information. This decoy could best provide the same interface routines as Important File.

6.3 Solving the Revocation Problem

One of the known difficulties with normal capability systems is known as the "revocation problem". The problem can be stated very simply: Once a capability has been granted to a user by the owner of an object, it cannot simply be deleted or modified by the owner if the object's owner later changes his mind. There have been many suggestions how to solve this problem[8] but these mostly create other problems or are unrealistically complex. The only "solution" of which I am aware which does not have such problems is that presented by Professor Roger Needham of Cambridge University, UK [17]. Put simply, he argued in the conference

[8] see for example https://www.usenix.org/legacy/publications/library/proceedings/fast03/tech/full_papers/aguilera/aguilera_html/nody are suspecte5.html

that capabilities should represent decisions taken earlier and should not imply that the object to which a capability refers should still exist. This is of course not a serious solution to the revocation problem.

As far as I am aware the only realistic and *simple* solution is that which Speedos bracket routines allow, as follows (see Figure 12).

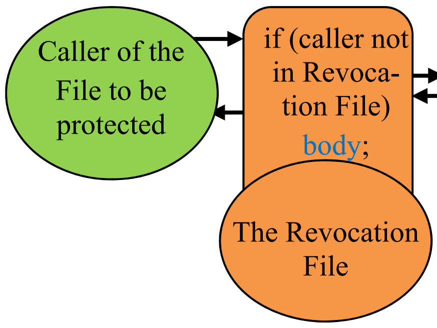
Caller of the File to be protected
if (caller not in Revocation File) body;
File to be protected
The Revocation File

Figure 12: Revoking Capabilities via a revocation list

The Revocation File contains a list of users whose capability for the file to be protected has been revoked. If a Caller attempts to use a capability which is on the list the call fails. The Revocation List might be organised by Unique Identifier of User, but a change list of access rights could be included if appropriate.

As an alternative a list of *permitted* uses could be used, in which case this is in effect an access control list.

6.4 Preventing the Release of Information

The aim in this case is to prevent the release of information from a module which its owner might regard as private or secret. In this case the best weapon against the theft of information (apart from careful management capabilities for the module) is the use of a qualifier with call-out brackets. These allow the owner of a file to examine and if appropriate change the parameters which are being passed from one module to another. Such parameters can only be passed in Speedos either by value or as module capabilities. They may *not* be passed as normal segment references within modules[9]. The two types of inter-module call parameters are kept on the kernel's call stack for each thread and can only be accessed (separately) in a very controlled way using special kernel controlled parameter registers. These are made accessible to call-out bracket routines, allowing them to be inspected and where appropriate modified.

Call-out brackets can, for example, ensure that the amount of information passed by value conforms to the purpose of the call and they can examine any module capabilities. If any capabilities are suspect the call-out bracket can invalidate them. How the bracket routine determines that capabilities being passed as parameters are suspicious can, for example, be determined from a list of permitted capability images, or (negatively) a list of forbidden capability images. Such a list could contain images of all the sensitive files which the owner (e.g. a company) regards as important. Normal semantic routines of the list module can be used to keep this list up-to-date. This solution is illustrated in Figure 13.

Alternatively the List might not be a Forbidden List but a Permitted List. Furthermore, the list might also contain entries identifying process threads which can be accepted, etc.

The designer of the security system can prevent the Called Module from receiving information simply by not issuing the call instruction.

[9] This restriction ensures that world-wide garbage collection is not necessary.

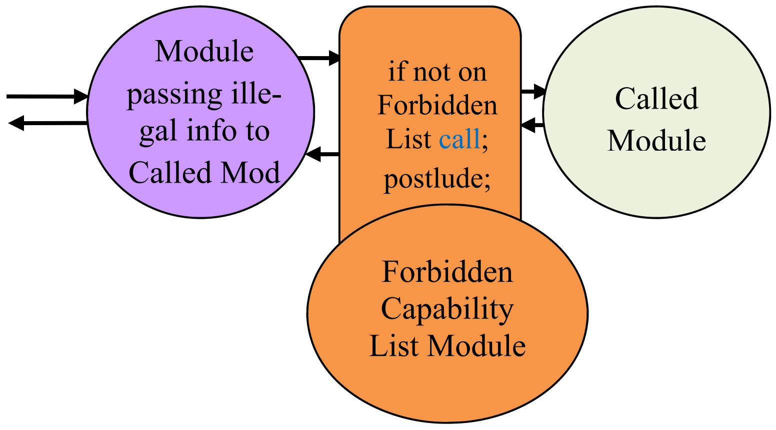
Module passing illegal info to Called Mod
if not on Forbidden List call; postlude;
Called Module
Forbidden Capability List Module

Figure 13: Preventing the release of information by using a Forbidden List

6.5 Encryption

The sending of messages via remote inter-module calls (RIMCs) is *always* used across the Internet in Speedos (see [16] volume 2 chapter 28, downloadable from the Speedos website (https://www.speedos-security.org/downloads.html). However, it is also possible to encrypt messages in a local environment by using qualifiers, as follows (see Figure 14):

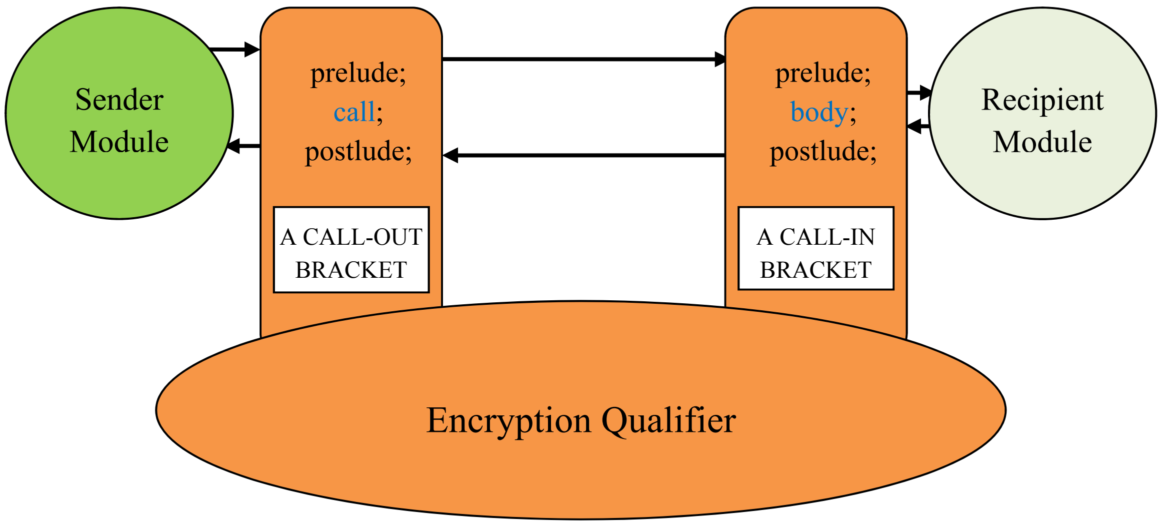

Figure 14: Encrypting and decrypting a local message

The call-out bracket of the sender encrypts the parameters destined for the recipient. The call-in bracket of the recipient then decrypts the same message, returning it to its original form. This may appear at first sight to be an case of unnecessary actionism, but in reality it serves a useful purpose, viz. it makes the sender's message unreadable to any further call-out bracket routines associated with the sender and to any call-in brackets associated with the recipient. Notice that the order in which bracket routines are executed is significant. This sequence is determined by the sender and by the recipient, as is explained in chapter 24 of volume 2 of "Making Computers Secure", downloadable from the Speedos website (https://www.speedos-security.org/downloads.html). An example of how such a situation can be programmed in Timor can be found in [13].

6.6 Ransomware

This section is inevitably more speculative, since I have no experience of ransomware as such. How this evil can be combatted depends in my view on exactly how an attack begins and proceeds. My impression is that in conventional systems a ransomware attack begins by the attacker (or his software) creating a new process before he can begin to encrypt important files. If this assumption is correct, the attacker will already stumble if he attempts to attack a Speedos system. The reason is that logging-in in Speedos does not involve creating a new process or thread but reactivating a thread which already exists in the Speedos persistent memory [10]. A brief description of the Speedos logging in and logging out mechanism appears on the Speedos website (https://www.speedos-security.org/logging-in-and-out.html) and more details can be found in volume 1 chapter 15 section 3 of the book "Making Computers Secure", which can be downloaded from the Speedos website (https://www.speedos-security.org/downloads.html).

This concept of persistent processes and threads allows users individually to protect their threads by activating a "logout" module immediately *before* logging out. When the kernel reactivates a persistent thread (as a user attempts to log in), this logout module (not the system kernel) immediately calls a further (user-owned) module which can communicate directly with the user to identify himself, using any technique which he chooses. He can change this method as often as he wishes (e.g. every day). There is no central file holding lists of users and their passwords and no standard logging in technique in contrast with conventional systems. Thus, provided that users take advantage of this mechanism and are sensitive to the reality that attacks can occur, ransomware attackers should have no chance of accessing a thread which would help them to carry out their evil work. Furthermore even if by a miracle they would not have access to the module capabilities which they wish to attack.

Finally I should add that there is *no* remote login mechanism[10].

Funding: This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

[10] Nevertheless a user can remotely access his files as if he were accessing them locally, provided that he has permission to access the remote computer *and* that he prepares a portable memory stick in an appropriate way (see volume 2 chapter 28 section 9 of the book "Making Computers Secure", downloadable from the Speedos website (https://www.speedos-security.org/downloads.html).

References

[1] R. S. Fabry, “Capability Based Addressing,” *Communications of the ACM,* vol. 17, no. 7, pp. 403-412, 1974.

[2] H. M. Levy, Capability-Based Computer Systems, Bedford, Mass.: Digital Press, 1984, p. 220.

[3] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, “HYDRA: The Kernel of a Multiprocessor Operating System,” *Communications of the ACM,* vol. 17, no. 3, pp. 336-345, 1974.

[4] A. K. Jones, “Capability Architecture Revisited,” *ACM Operating Systems Review,* vol. 14, no. 3, pp. 33-35, 1980.

[5] J. B. Dennis and E. C. Van Horn, “Programming Semantics for Multiprogrammed Computations,” *Communications of the ACM,* vol. 9, no. 3, pp. 143-155, 1966.

[6] J. L. Keedy, “Paging and Small Segments: A Memory Management Model,” *Proceedings of the 8th World Computer Congress, Melbourne, Australia,* pp. 337-342, 1980.

[7] D. L. Parnas, “Information Distribution Aspects of Design Methodology,” in *Proceedings of the 5th World Computer Congress*, 1971.

[8] D. L. Parnas, “A Technique for Module Specification with Examples,” *Communications of the ACM,* vol. 15, no. 5, pp. 330-336, 1972.

[9] D. L. Parnas, “On the Criteria to be Used in Decomposing Systems into Modules,” *Communications of the ACM,* vol. 15, no. 12, pp. 1053-1058, 1972.

[10] J. L. Keedy, “Persistent Programming with Speedos and Timor,” in *SPEEDOS Website (https://www.speedos-security.org/)*, 2024.

[11] J. L. Keedy, “S-RISC: Adding Security to RISC Computers,” SPEEDOS Website (https://www.speedos-security.org/), 2023.

[12] J. L. Keedy, M. Evered, A. Schmolitzky and G. Menger, “Attribute Types and Bracket Implementations,” in *25th International Conference on Technology of Object Oriented Systems, TOOLS 25*, Melbourne, 1997.

[13] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, “Call-out Bracket Methods in Timor,” *Journal of Object Technology,* vol. 5, no. 1, pp. 51-67, 2006.

[14] B. W. Lampson, “Protection,” *ACM Operating Systems Review,* vol. 8, no. 1, pp. 18-24, January 1974.

[15] K. Espenlaub, Design of the SPEEDOS Operating System Kernel, Ulm, Germany: Ph.D. thesis, The University of Ulm, Department of Computer Structures, Computer Science Faculty, 2005.

[16] J. L. Keedy, Making Computers Secure, Speedos Website, https://www.speedos-security.org/, 2021.

[17] R. M. Needham, “Capabilities and Security,” in *Security and Persistence, International Workshop on Computer Architectures to Support Security and Persistence of Information*, Bremen, Germany, 1990.