ACLs don't

Tyler Close
Hewlett-Packard Labs
Palo Alto, CA
Email: tyler.close@hp.com

Abstract

The ACL model is unable to make correct access decisions for interactions involving more than two principals, since required information is not retained across message sends. Though this deficiency has long been documented in the published literature, it is not widely understood. This logic error in the ACL model is exploited by both the clickjacking and Cross-Site Request Forgery attacks that affect many Web applications.

1. Introduction

In the last few years, increasing attention has been devoted to attacks which are distinctly different in nature from the major vulnerabilities discussed in the past. Previously, buffer overflow, SQL injection and Cross-Site Scripting (XSS) garnered the most attention. These attacks all share a common modus operandi in that they inject code into a victim program and thus make it behave according to the attacker's wishes. For example, in a buffer overflow attack, data provided by the attacker is written to a buffer of dimensions declared by the victim. The attacker provides more data than fits in the buffer, causing data to be written into another region of memory, changing the logic of the victim program. In an SQL injection attack, a literal value provided by the attacker is included verbatim in an SQL expression constructed by the victim program. The literal value contains text that matches the SQL syntax for closing a literal expression, followed by more SQL expressions. When the constructed SQL query is executed, the attacker gains unexpected access to the victim's database. Similarly, an XSS attack involves attacker input that breaks out of the intended quoting context in HTML, CSS, JavaScript or URL syntax.

Recent attacks such as Cross-Site Request Forgery (CSRF) and clickjacking don't fit this familiar mold. Neither attack requires injecting code chosen by the attacker into the victim program. Instead, both attacks use the victim's existing program logic to unexpected ends. Using messages that follow the syntactic conventions expected for legitimate requests, the attacker makes use of resources that should be inaccessible according to the application's access policy. For example, in a CSRF attack, an attacker may successfully send a buy request to a victim's stock trading application, though no one but the account's owner is supposed to be allowed to do that. In a clickjacking attack, an attacker may start a web-cam recording, though no one but the computer's user is supposed to be allowed to do that. Though these victim applications may correctly implement traditional access control lists (ACLs), somehow the attacker still gains access to resources that are supposed to be inaccessible, and does so without modifying any of the application's program logic.

Stranger still, the popular counter-measures for these attacks do not involve fixing incorrect ACL configurations, nor adding ACLs to unprotected parts of the application. Though these attacks circumvent the application's access policy, ACLs seem to play no part in fixing the problems. Why do ACLs seem to be so ineffective at fulfilling their basic purpose of controlling access?

This paper provides an explanation of the common modus operandi of the new wave of attacks that includes CSRF and clickjacking. This explanation clarifies how these attacks exploit flaws in the ACL model. These flaws render the ACL model ineffective for access control in scenarios involving more than two principals, such as are common on the Web.

The first part of this paper starts with the fundamentals of the access matrix, explaining how it operates in multi-party scenarios. This explanation points out exactly where the logic errors occur in the ACL model and what their effects are. Enhancements of the ACL model and alternate models are discussed in terms of

how they may address these logic errors. The middle part of this paper then explains how these errors in the ACL model are manifested in contemporary systems, like the Web. Finally, the paper closes with a discussion of related works that have discussed these problems.

2. Access Matrix

The 1971 paper "Protection" [9], introduced the access matrix as a way to model the permissions in a software system and how they may be shared and exercised. An access matrix is a table where each row is labeled with the identifier for a principal, who may send messages, and each column is labeled with the identifier for a protected object, which may be a subject of messages. A principal is considered to be a kind of protected object and so may appear as both a row and column. Each cell in an access matrix contains entries identifying the operations the corresponding principal is permitted to perform on the corresponding object. For example, Table 1 is an access matrix of two principals and three objects. The Vendor principal has read/write permission to the log.txt file. The User has read/write permission to two files: main.c and a.out. Entry $A_{R,c,p}$ refers to the permission p in row R, column c of access matrix A. For example, entry $A_{U,a,write}$ refers to the User's write permission to the a.out file.

The Protection paper describes two implementation techniques for storing the access matrix: the capability and the access control list (ACL). Performance tradeoffs are the only considerations presented for choosing between these techniques.

This section examines the ACL and the capability for semantic differences. The behavior of each implementation is detailed for an identical scenario involving three principals, one of which is a software agent instantiated to mediate an interaction between the other two principals. The software agent is a Compiler which compiles source code provided by the User, and maintains usage statistics for the Vendor. The introduction of the Protection paper proposes this exact scenario as a useful way to evaluate an access control mechanism. As will be discussed, the scenario was again taken up in a later paper on access control. [1]

1. For some readers, the particulars of this scenario may seem a little dated. In that case, wherever the text talks of the Vendor, think Web site; for Compiler, think Web browser; and for User, think Web page. This correspondence is explained in greater depth in the next section of this paper. After reading some of this section, you may wish to skip ahead to the subsequent section and then come back.

Table 1. Access matrix for file access

	main.c (m)	a.out (a)	log.txt (l)
Vendor (V)			read, write
User (U)	read, write	read, write	

2.1. ACL checking

In the ACL model, a message consists of an identification of the sending process followed by an arbitrary amount of data. The identification is provided by the system and therefore cannot be forged. Upon receipt of a message, a reference monitor uses the sender identification and the message data to check that corresponding entries in the access matrix specify the required permissions. For example, starting from the access matrix in Table 1, a User compiles a source code file, outputting the results to a local output file.

The interaction begins with the User executing a Vendor provided compiler program. This execution adds a new row and column to the access matrix for the running Compiler instance. [2] At this point, the User has 'call' permission on the Compiler and the Compiler has 'write' permission on the log.txt file. To set up the compilation, the User grants the Compiler at least 'read' permission to main.c and 'write' permission to a.out, resulting in the access matrix in Table 2.

To initiate compilation, the User sends a compile message to the Compiler, specifying the names of the input source code file and the output object code file.

User:

```
Compiler.compile("main.c",
                 "a.out")
```

Upon receipt of the compile message, the Compiler's reference monitor uses the message sender identifier to check that the message sender has 'call' permission on the Compiler. This check searches down column A_c of the access matrix, finding entry $A_{U,c,call}$, and so the access check passes and computation proceeds. The Compiler then reads the input source code by sending a read message to the Filesystem, specifying the name of the file to read.

Compiler:

```
Filesystem.read("main.c")
```

Upon receipt of the read message, the Filesystem's reference monitor checks that the message sender has

2. Today's mainstream operating systems typically do not use a separate principal for a running process, instead running the process as the User. To create a multi-party scenario, the current example assumes this violation of least privilege is not made. This assumption was also made in the original presentation in the Protection paper.

Table 2. Access matrix for an instance of the compiler program

	Compiler (c)	main.c (m)	a.out (a)	log.txt (l)
Vendor (V)				read, write
User (U)	call	read, write	read, write	
Compiler (C)		read	write	write

'read' permission on the specified file. This check finds entry $A_{C,m,read}$ of the access matrix, and so the check passes and the file contents are returned.

To maintain usage information, the Compiler then sends an append message to the Filesystem, specifying the name of the Vendor's log file.

Compiler:

```
Filesystem.append("log.txt",
                  "log entry")
```

Entry $A_{C,l,write}$ is checked before writing the log entry.

Finally, the Compiler sends a write message to output the object code.

Compiler:

```
Filesystem.write("a.out",
                 "compiled code")
```

Entry $A_{C,a,write}$ authorizes this output.

2.2. Capability transfer

In the capability model, a message consists of a list of permissions and an arbitrary amount of data. Each permission in the list is selected by the message sender, choosing from its held permissions. The selected permissions are added to the message by the system and therefore cannot be forged. A message recipient can in turn send messages using any of the permissions provided by the message, or already held by the recipient. Once again using the same example, and starting from the access matrix in Table 1, a User compiles a source code file, outputting the results to a local output file.

Again, the interaction begins with the User executing a Vendor provided compiler program. This execution adds a new row and column to the access matrix for the running Compiler instance. At this point, the User has 'call' permission on the Compiler and the Compiler has 'write' permission on the log.txt file.

To initiate compilation, the User sends a compile message to the Compiler, specifying permission to the input source code file and permission to the output object code file. The User's reference monitor constructs this message, copying the specified permissions from row A_U of the access matrix.

User:

$A_{U,c,call}$`.compile(`$A_{U,m,read}$`,` $A_{U,a,write}$`)`

Upon receipt of the compile message by the Compiler, the access matrix can be construed as again being in the state depicted in Table 2, though it's more accurate to think of the Compiler as directly wielding the permissions received from the User, rather than having its own independent permissions. For example, using the permission received in the compile message, the Compiler's message to read the input source code takes the form:

Compiler:

$A_{U,m,read}$`.read()`

To maintain usage information, the Compiler sends an append message using the permission provided by the Vendor.

Compiler:

$A_{V,l,write}$`.append("log entry")`

Like with the 'read' permission provided by the compile message, the Compiler holds a copy of the 'write' permission the Vendor provided when constructing the Compiler.

Finally, the Compiler sends a write message to output the object code.

Compiler:

$A_{U,a,write}$`.write("compiled code")`

2.3. Confused Deputy attack

For the worked example, both ACL checking and capability transfer yield the same access decisions, permitting each step in the example to proceed; however, the two differ in the process used to reach these decisions. Consider the final access decision, which authorizes the output of the compiled code. In capability transfer, the reference monitor determined the exercised permission, $A_{U,a,write}$, by performing a look-up against row A_U at the time the compile message was sent. In ACL checking, the reference monitor determined the exercised permission, $A_{C,a,write}$, by performing a look-up against column A_a, at the time the write message was sent. Exercising $A_{U,a,write}$ or $A_{C,a,write}$ has the same effect, so this distinction makes no difference, in this case. In a similar case, an attacker could use this distinction to cause a significant difference.

For ACL checking, consider the case where the User is an attacker and so where before the User sent the compile message:

User:

```
Compiler.compile("main.c",
                 "a.out")
```

, instead the User now sends the compile message:

User:

```
Compiler.compile("main.c",
                 "log.txt")
```

The text string `"log.txt"`, sent by the User, becomes the value of the identifier provided by the Compiler in its message to output the compiled code:

Compiler:

```
Filesystem.write("log.txt",
                 "compiled code")
```

This time, the reference monitor performs a look-up against column A_l of the access matrix and so entry $A_{C,l,write}$ authorizes the message. Consequently, the Vendor's usage log is overwritten with compiled code.

In capability transfer, the User's attempt to construct a compile message specifying write permission to log.txt is rejected by the reference monitor, since the look-up against row A_U of the access matrix shows the User does not possess this permission.

In the original presentation of this attack [8], the Compiler is termed a Confused Deputy. The Compiler has been deputized by the Vendor to operate on his behalf, but also operates on behalf of the User. Though it has the responsibility of mediating between distinct parties, the Compiler does not have a mechanism for keeping separate the authority received from these different sources. The implicit expectation for the write message is that permission received from the User will be exercised, but the Compiler has no way to express this expectation and so permission received from the Vendor is exercised instead. The Vendor contributed write permission is confused for one contributed by the User; hence, the Confused Deputy. [3]

2.4. ACLs don't authorize correctly

The intent for the compile method is to output the compiled code to one of the User's files. In capability transfer, the access check is performed as soon as the User selects an output file and is carried through the rest of the computation in the form of a capability. In ACL checking, only a data string is produced by the User's selection of a file and the access check is delayed until the output file is about to be written. At this late stage, the ACL reference monitor does not know that the value of the file identifier is chosen by the User, not the Compiler, so the access check yields an incorrect decision.

The paper "Authentication in Distributed Systems: Theory and Practice" by Lampson et al (Speaks-for) [10] begins with a summary of the access matrix model presented in Protection [9]. This summary defines the inputs to the ACL reference monitor:

> The reference monitor bases its decision on the principal making the request, the operation in the request, and an access rule that controls which principals may perform that operation on the object.

Given these inputs, the ACL reference monitor is unable to produce correct access decisions for scenarios involving more than two principals, since the particulars of the operation may have been determined by a principal other than the request sender or the request receiver, and this information is not available to the reference monitor.

In contrast to the ACL reference monitor, the capability reference monitor performs access checks earlier in the call chain of messages, when the principal designating a particular object is still known. The result of an access check is reified as a capability that can be transferred to other principals, and so used in messages that combine the permission with those of the other principals. A message at the end of such a call chain may exercise permissions contributed by many principals, each one authorizing some specific, smaller part of the requested operation. In a capability language, this construction of messages from capabilities is expressed using the language's normal argument passing syntax.

When viewed in this way, it is clear that the ACL model and the capability model are fundamentally different ways of modeling the authorization of requests. Although both models can be construed as algorithms operating on an access matrix, they use different parts of the matrix at different times and grow the matrix in different ways, resulting in different access decisions for identical scenarios. Therefore, the view presented in the Protection paper that ACLs and capabilities are merely different implementation choices for a single access model embodied by the access matrix is incorrect. Moreover, for a given access policy, access

3. The shorthand for this attack, 'Confused Deputy', is a little unfortunate since it implies some lack of competence on the part of the deputy software agent. As is clarified in this section, the confusion of one permission with another is a condition created by the ACL model and which the deputy may be unable to rectify.

decisions are not results that can benignly differ: one is right, the other wrong.

2.4.1. Role-based access control, etc. The Speaks-for paper [10] begins with the statement:

> Most computer security uses the access control model [9], which provides a basis for secrecy and integrity security policies.

As is shown in this paper, the provided basis is inadequate for multi-party scenarios. Unfortunately, the first clause in the quote is accurate; the ACL model, sometimes called Identity-based Access Control (IBAC), is pervasive in computer security. Variations on this model, such as Role-based Access Control (RBAC) [5] and Attribute-based Access Control (ABAC) [1], which seek to address the burdens of client identification, are still vulnerable to the Confused Deputy attack discussed in this paper. These variations on the IBAC model also suffer from the same problem of delaying the access control check until a late stage, when information needed to make a correct access decision is no longer available. Just as with IBAC, RBAC and ABAC fail to account for effects on the requested operation by principals other than the immediate message sender.

2.4.2. setuid. In the compilation scenario, the ACL reference monitor uses the Compiler's identity when performing the access check for the write message that outputs the compiled code. The Confused Deputy attack exploits the fact that the Compiler has additional permissions, such as write access to the log.txt file, beyond what the User possesses. It is tempting to believe these attacks can be addressed by switching the principal identifier for a running program, such as can be done with the setuid() command in Unix.

In the compilation scenario, none of the existing principals is an appropriate one for execution of the output message. As has been shown, running as the Compiler results in a Confused Deputy attack. The Vendor does not have write permission to the User's output file. The compiler should not be allowed to run as the User, since the User may have access to files that should be protected from the Compiler. The need for this constraint is more obvious in an environment where the program's source code is not available to be inspected by the User. For example, this is the case in a distributed computing scenario, where the Compiler might be a service running on a remote server.

Essentially, none of the divisions of permission embodied by the principals correctly isolates the permission needed for the write message.

It is also noteworthy that this technique depends upon the rigorous participation of the application programmer to make the uid setting system calls. Consequently, the correct implementation of an access policy cannot be ascertained by an examination of the ACLs configured for an application, but must also include an examination of the program's source code. To date, this technique has been error prone [2].

2.4.3. Stack introspection. Since principal identities are too coarse grained to segment authority received from separate sources, perhaps an intersection of identities could provide the needed specificity. Stack introspection, as implemented in the Java platform [6], bases access decisions on the intersection of the permissions held by a list of principals. By default, this list of principals includes each caller in the call chain leading up to an access check. The access is only allowed if every principal in the list possesses the required permission.

If the value of the object identifier used in an access check was determined solely by principals represented in the call chain, this technique defends against a Confused Deputy attack. Since each principal has permission to perform the operation on its own, none of the principals can increase their authority by having another principal execute the operation on their behalf. On the other hand, if a principal not represented in the call chain could have an effect on the value of the object identifier, the Confused Deputy vulnerability remains. For example, if any of the callers computed the identifier value based on the value of state held in their lexical scope, such as by reading an object member field, there may be principals who could affect the value of the identifier without being in the call chain. These principals were in the call chain when the member field was assigned, but aren't when the field is later read.

To address this problem, the Java API provides a means to manually track principals across call chains. At the time a member field is assigned, the caller can take a snapshot of the list of principals in the current call chain [4]. Later, a caller can switch from the default call chain based principal list to the saved snapshot list. By switching lists, principals who were represented in the current call chain, but not in the saved list, are no longer considered in any access check. Consequently, effects these principals may have had on the object identifier used in an access check could again lead to Confused Deputy vulnerability. It is unclear if the Java API provides a way to merge the current call chain list

4. java.security.AccessController.getContext()

with a saved list, but presumably such functionality could be provided. In either event, this model introduces a need to maintain a corresponding principal list for every member field, and to keep this list consistent with the sequence of assignments done to the member field. If ever a principal has an effect on the value of a member field, without being represented in the corresponding principal list, there is an opportunity for a Confused Deputy attack.

There are also additional opportunities for Confused Deputy in a stack introspection design. The purpose of a software agent like the Compiler is to mediate an interaction between two or more other principals. Often this mediation involves using the union of their permissions. For example, consider an operation of two parameters. The software agent is to use an object specified by principal A as the first argument and an object specified by principal B as the second argument. Principal A should be prohibited from using principal B's object as the first argument, and vice-versa. In this case, no single principal list can fully express these access constraints, so access checks done by the operation's implementation are necessarily vulnerable to Confused Deputy attack.

By providing an indivisible representation of an access matrix entry, a capability essentially enforces the discussed tracking of the authorizing principal for every object identifier held by a caller. Since the representation is indivisible, any principal who had an effect on the object identifier must also have held the corresponding permission. Under stack introspection, this tracking of principals is manually implemented at the discretion of the application programmer. Since many capabilities can be provided as arguments to an operation, independently tracked authorization chains can correctly authorize multi-argument operations. Such access decisions cannot be correctly done using stack introspection, since at best the model supports tracking of a single authorization chain per operation.

2.5. ACLs don't authenticate reliably

One of the core features of an ACL message system is the provision of an identifier on every message that identifies the sender. The ability to know "Who said this?" for any given message is generally thought to be important and useful information, and so many deployed systems provide this client authentication. As has already been shown, client authentication is actually misleading when used as the input to an access decision. Other message recipient routines may also rely on client authentication for purposes for which it is unreliable.

In the compilation scenario, the User can provide any object identifier as the second argument in the compile message to the Compiler. The Compiler necessarily uses this argument as the target for the write message sent to output the compiled code. Since this message target is determined by the User, the User can cause the Compiler to send a write message to any target object of the User's choosing. Depending on the particular implementation, the target object may be limited to one of type file in this case; however, in other implementations and in general, this type restriction may not apply. Consequently, it should in general be assumed that a first principal that can call a second principal can cause that second principal to send a message to any target object of the first principal's choosing. Therefore, a message recipient must not assume that the mere sending of a message represents an expression of intent by the message sender. The targeting of a message may not be something that the message sender exerts control over. Since no intent can be associated with the sender identifier provided by a message, the ways in which a message recipient can rely on this client authentication are quite restricted. These limitations of client authentication may not be well understood by many system designers and application developers.

For example, consider a recipient that presents the contents of a received message as being authored by the message sender, as an electronic bulletin board might do. While it may be true that the bits in the message passed through the sender's computer, these bits may not represent an expression by the sender. In the case of the Compiler, the content of a write message is largely determined by the User; so attributing this content to the Compiler is incorrect. The Compiler's intent is merely to place output where the User directed; not to claim authorship of any expression in that content. This logic error is especially dangerous in an environment where a software agent doesn't run as a distinct principal, but rather under the identity of one of its human users, such as is the case for all mainstream operating systems. In the compilation scenario, neither the User nor the Vendor can sensibly be identified as author of the compilation output. As discussed, the Vendor has no intent to claim authorship of the compilation output. The User lacks control over messages sent by the Compiler, since the Compiler's code is provided by the Vendor.

In general, a security reviewer should approach any use of client authentication in a software system with suspicion. Little can be reliably concluded based on client authentication.

2.6. ACLs don't assign accountability correctly

In a Confused Deputy attack, the deputy's permissions are exercised in a way the deputy did not intend and may have been unable to prevent. For example, in the compilation scenario, the User provides the impetus for overwriting the log.txt file, not the Compiler. Holding the Compiler accountable [5] for this abuse is neither fair, nor useful. Redress of the situation requires identification of the User, not the Compiler.

For a principal to be usefully held accountable for a message, that principal must have had intent associated with that message. The previous section showed the client authentication attached to a message in an ACL system does not provide a reliable indication of intent. Consequently, this authentication is not a reliable means of assigning accountability for messages. In an attack scenario, accountability is incorrectly assigned to a principal who was merely forwarding a message as directed, and required to provide basic functionality.

Interestingly, ACLs also fail to assign accountability correctly in legitimate cases. For example, in the non-attack case of the compilation scenario, the Compiler is held accountable for the write to the a.out file. Again the Compiler is acting at the impetus of the User; and this time is also using permission received from the User. Holding the Compiler accountable for this write operation makes little sense, but the ACL model provides no alternative. The write message only identifies the Compiler. The ACL reference monitor does not know what messages instigated sending of the write message. The access matrix does not record the source of permissions held by the Compiler.

In the case of a software agent, like the Compiler, it may be possible to maintain a separate log of what delegation of permission was done, what messages were sent and so through analysis of the software agent's source code, determine a more appropriate principal to hold accountable for an operation. This forensic task is more difficult when the acting principal is a human user, rather than a software agent. For example, consider a case where a co-author has been granted write permission to a document file created by a first author. The co-author then further delegates this write permission to a student, telling the student to submit a homework assignment to this location. When the student dutifully submits the assignment, the document file is overwritten with unrelated information. In this scenario, the student is acting in good faith, like the Compiler in the compilation scenario, and is merely forwarding output as directed. Though the student actually performed the act of overwriting the document file, the document's first author should rightly hold the co-author accountable for this act, not the student. Under the ACL model, the write message only identifies the student; and the access matrix, even if augmented with additional logging, doesn't provide sufficient means to determine that the co-author should be held accountable for this act. Such a determination requires knowing what the co-author said to the student.

In contrast to the ACL model, the capability model doesn't perform delegation by providing the delegate with its own unique representation of a permission. Instead, the delegate directly manipulates the same capability held by the principal that performed the delegation. For example, in the current scenario the first author would create a capability to a file. This capability would then be passed to the co-author. The co-author then passes the same capability to the student. When the student exercises the capability, a simple equality test shows it to be the same one created by the first author. Accountability for the write operation is therefore assigned to the first author. Again through a simple equality test, the first author can determine that the exercised capability is the one delegated to the co-author and so in turn blame the co-author. The co-author, having sent the capability to the student under false pretenses, is ill-equipped to further pass the buck. Regardless of whether or not the co-author in turn blames someone else, the first author has collected sufficient information to know that delegation to the co-author ultimately resulted in unwanted operations and so take action by revoking [6] the capability and not granting the co-author write access in future.

The discussed scenario involves a chain of delegations: from the first author to the co-author, and then from the co-author to the student. In the ACL model, accountability for a message is assigned to the principal at the end of this delegation chain. In the capability model, accountability is assigned to the principal at the start of the delegation chain. Working the delegation chain backwards to a point where redress action can be taken is difficult and sometimes impossible. Working down the delegation chain from start to finish is a feasible way for an injured party to determine an appropriate redress action.

In richer multi-party interactions, a principal may over time be introduced to new principals. Some of

5. The term 'accountability' is used here in the same sense as it is used in [15].

6. The revocation of this capability can be done using the Caretaker pattern described in [12].

these introductions may come from distinct principals. Consequently, a principal has an evolving understanding of other principals and relationships between them. The Horton capability protocol [13] provides a comprehensive way to track delegations, and take appropriate redress action, in such dynamic multi-party scenarios.

2.7. Avoiding Confused Deputy within a capability application

Although the capability model itself is not vulnerable to Confused Deputy attacks, an application built for a capability system could make itself vulnerable to a Confused Deputy attack by effectively re-implementing an ACL design on top of capabilities. For example, an application that keeps a mapping from string names to capabilities and communicates with its clients in terms of these string names is effectively re-implementing the ACL model and so makes itself vulnerable to Confused Deputy attacks.

The crucial step in a Confused Deputy attack occurs when an object identifier passes through an intermediate principal without being checked against the access matrix. For example, in the compilation scenario, a file identifier passes through the Compiler on its way from the User to the Filesystem. Once the file identifier has passed from the User to the Compiler, information needed to perform a correct access check has been lost. This isolation of the critical error suggests that Confused Deputy attacks can be discovered, and fixed, through an analysis of the API for messages between protection domains.

Wherever a message parameter is of a raw data type, like a text string or integer, and its value identifies an object for which access permission is required, there is a possibility of a Confused Deputy attack. Determining whether or not there is an attack requires examination of the inputs to the routine that eventually performs the dereference operation of looking up the corresponding permission for a given identifier. In a capability system, one of the inputs to this dereference routine must be the list of all permissions that an identifier could map to. If all principals who can determine the value of the identifier should have the ability to exercise any of the permissions in the list, there is no Confused Deputy vulnerability. Otherwise, the application may be vulnerable to a Confused Deputy attack, which can be constructed by identifying a principal who can inject the unexpected identifier but should not be able to exercise the corresponding permission.

Sometimes, this attack may be thwarted by an intermediary that performs ad-hoc tests against the value of an identifier before passing it along. For example, this operation is commonly performed by some HTTP firewalls that reject HTTP requests having a Request-URI with a disallowed value. This operation has the effect of making it seem like the list of permissions used by the HTTP server's dereference routine is shorter than it actually is. In cases where a Confused Deputy attack can be defended against in this way, it is more robust to instead actually shorten the list of permissions used by the deference routine. For example, the HTTP server may be configured to only have read access to those files that all Web users should have read access to. In this case, ad-hoc checking of the Request-URI is unnecessary.

In other cases, it may be that permissions in the list should not be uniformly accessible to all clients. For example, in the compilation scenario each user has access to a different set of files. In such cases, it is best to modify the API to use the corresponding capability wherever the object identifier is used, such as was shown for the capability version of the compilation scenario. More concretely, and assuming a Unix-like system, the compiler API should be modified to use file descriptors, instead of filenames. Before calling the Compiler, the User opens an output file and includes the file descriptor in the compile message, in place of the output filename. The input filename and log filename should similarly be replaced with corresponding file descriptors.

3. Contemporary examples

Since Confused Deputy is an attack on the ACL model itself, we should expect to find instances of this attack in any system using the ACL model for interactions involving more than two principals. Systems using the ACL model are widespread, making the restriction to multi-party interactions the more limiting factor. Distributed computing systems that support interaction between multiple principals have been slow to deployment, but the Web finally brought this functionality into the mainstream. Unfortunately, the Web has, to date, mostly done so using the ACL model and so exhibits the expected Confused Deputy vulnerabilities.

The previous section describes a scenario in which an HTTP server is a Confused Deputy, operating with filesystem permissions from a Web site operator, but acting on requests from Web surfers. The Web browser also provides many opportunities for Confused Deputy attacks, as it is deputized with HTTP cookies by one Web site, while rendering Web pages from other Web sites.

3.1. Cross-Site Request Forgery

For example, consider an investment account at a stock broker's Web site [7]. This Web application provides a feature to buy stocks. The resource to make a purchase is located at a URL like: `https://example.com/buy.php`. A POST request sent to this resource provides the stock ticker and number of shares to buy. In the planned legitimate use-case, this resource is invoked from a FORM in an HTML page served by the stock broker's Web site. In an attack scenario, another Web site could serve an HTML page that contains an identical FORM, prepopulated with a ticker symbol and number of shares. Using JavaScript, this FORM can be automatically submitted on page load. Assuming the user is currently logged into the stock broker's Web site, the user's browser will send the POST request to the stock purchase resource and include any cookies, or HTTP authentication credentials, set up with the stock broker's Web site. The stock broker's Web site receives a POST request containing exactly the same Request-URI, entity body and cookies as in the normal case. The user may see nothing out of the ordinary in the Web browser's presentation.

The above attack is analogous to the previously studied compilation scenario. Table 3 lists the corresponding elements in each attack. In the browser based attack, the ACL reference monitor executes on the stock broker's Web server, using the HTTP cookie and URL to look-up an entry in the application's access matrix.

When popularized, this attack was given the name Cross-Site Request Forgery (CSRF) [16]. The use of the term forgery is not ideal, since no signature or other authenticity marker is altered or imitated. Instead the problem is the Confused Deputy vulnerability inherent in the ACL model.

To defend against this attack, the CSRF paper recommends use of an unguessable token in the HTML FORM served by the stock broker's Web site. The stock purchase resource then checks that a received POST request contains the expected token. The browser's Same Origin Policy prevents an attack page from extracting a token from a legitimate page. This technique has become the most popular defense to CSRF vulnerability.

Although the CSRF article makes no reference to the Confused Deputy attack or capabilities, the suggested defense is effectively to transition the application away from the ACL model and to the capability model. Comparing the suggested defense to the capability-based solution for the compilation scenario, and again assuming a Unix-like system: the URL is like the filename; and the unguessable token is like a file descriptor, approximating the unforgeability of a capability with unguessability. A legitimate page from the stock broker's Web site first *opens* the stock purchase resource, receiving an unguessable secret. The legitimate page then uses this unguessable secret when instructing the browser to *write* to the stock purchase resource.

7. This example is adapted from the one presented in [16].

3.2. Clickjacking

In the example of the previous section, the use of an unguessable token transitions the POST request that makes a stock purchase into the capability model; however, the GET request to set up a purchase remains in the ACL model. Recently, it has been shown how this setup phase remains vulnerable to a Confused Deputy attack.

For example, the investment account at a stock broker's Web site also includes a feature to sell owned shares. In the portfolio summary page, beside each holding, is a button labeled "close position". Clicking this button sells the held shares. The corresponding HTML FORM may or may not be protected against CSRF using the previously discussed technique. The portfolio summary page is located at a URL like: `https://example.com/home.php`. An attacker's page, served from another Web site, can include an HTML IFRAME that references the portfolio summary page. An IFRAME creates an inline child window that displays a referenced page. Using Cascading Style Sheets (CSS), the attacker's page can style the IFRAME to have no border and be transparent. To the user, this IFRAME is completely invisible. Underneath the IFRAME, the attacker's page puts content that entices the user to click at a specific location. For example, this content could be a "punch the monkey" type game, or just a link "click here for free stuff". The attacker has positioned this click target so that it is directly underneath the "close position" button in the invisible IFRAME. When the user clicks, the click is delivered to the invisible button since it is on top of the attacker's content.

The particular rendering techniques highlighted by the clickjacking research [7] are fascinating, but this glitz may distract from the actual underlying problem. Much simpler techniques could also be used in a plausible clickjacking attack, and so perhaps make the actual problem more apparent. For example, consider a scenario where a single button press can launch a

Table 3. Corresponding elements in Confused Deputy attacks

element	compilation scenario	stock purchase scenario (CSRF)	stock sale scenario (clickjacking)
Confused Deputy	Compiler	Web browser	Web browser
message sender identifier	process UID	HTTP cookie	HTTP cookie
victim	Vendor	stock broker application	stock broker application
attacker	User	visited HTML page	visited HTML page
unexpected object identifier	`"log.txt"`	`https://example.com/buy.php`	`https://example.com/home.php`
abused object	log.txt file	stock purchase resource	account summary page
operation	write	POST	GET

dangerous action. The attacker could engage the user in a game of mouse clicking, and then, just as the user was about to click, navigate the browser to the page containing the privileged button. In this case, there are no IFRAMEs or transparency settings, just a simple page navigation. Even the use of JavaScript is unnecessary as the navigation could be triggered by the immediately preceding mouse click. Stripping down the scenario to this extent shows that the attacker can cause mischief using only the authority to link to a private page.

An HTML link is a request for the browser to place named content at a specified on-screen location. When the browser includes cookies in the GET request to fetch the content, it is acting as a Confused Deputy. Like in the compilation scenario, the requestor does not have permission to access the named resource, but can provide the resource's name to the deputy, who will access the resource on the requestor's behalf. In clickjacking, the requestor is the creator of the HTML link and the deputy is again the Web browser. A full listing of the corresponding elements in the attacks is shown in Table 3. This formulation of a Confused Deputy attack is quite similar to the previously discussed CSRF attack. In that attack, the attacker causes a POST request to a victim site, accompanied by the victim site's cookies. Clickjacking can similarly be thought of as an attack in which the attacker causes a GET request to a victim site, accompanied by the victim site's cookies. In a CSRF attack, the payoff to the attacker comes from the side-effects of the POST request. In a clickjacking attack, the payoff comes from the on-screen positioning of private controls. Gratification is slightly delayed in the clickjacking attack, since it doesn't come until the user clicks, but the subterfuge comes before the final click, in the set up of the click target.

A complete defense against both clickjacking and CSRF can be created by completing the transition away from the ACL model and to the capability model. If the URL for a page containing privileged buttons included an unguessable secret, the attacker would be unable to create an IFRAME element that refers to the page. Similarly, the attacker would also be unable to navigate the browser to that page at an unexpected moment. By taking away the attacker's ability to display the privileged buttons, we take away the opportunity to play tricks with the timing and positioning of their display.

In the capability model, the rendering tricks used in the clickjacking attacks are not dangerous and so need not be restricted. For example, there's no need to place restrictions on the creation of IFRAMEs or opacity styling. If an attacker doesn't know the unguessable URL for a victim page, he is unable to load the page and so is unable to trick the user into interacting with the page. If a legitimate site does know the unguessable URL for a page at a partner site, it can load the page and customize its presentation. Such customization isn't trickery, since there's no need for trickery. The legitimate site can already send any request it likes using the unguessable URL; interaction from the user isn't needed.

3.2.1. web-key. A surprising number of the problems with today's Web are directly attributable to the use of the ACL model. The web-key paper [4] explains many of these problems and also describes how best to use unguessable URLs to address these problems and move the Web to the capability model. No changes to Web protocols, formats, user agents or server-side infrastructure are required to make this transition. The required changes are limited to the URL namespace defined by a Web application.

3.3. Click fraud

Both the Cross-Site Request Forgery and clickjacking attacks target the misuse of client authentication as an input to an access decision. As discussed earlier in section 2.5, other message recipient routines may also rely on client authentication for purposes for which it is unreliable.

A Web browser only sends two kinds of requests: GET and POST. Any visited Web page can cause the browser to send both GET and POST requests to any URL known to the Web page. An arbitrary GET request can be sent using the HTML IMG element. An arbitrary POST request can be sent using the FORM element, and using JavaScript to programmatically submit the FORM on page load. The web browser exerts no control over the target of a request. As discussed previously, this is normal and necessary behavior for a software agent in a messaging system. Consequently, the server-side of a Web application should not associate any user intent with any received request, based on client authentication.

For example, in pay-per-click online advertising, an advertiser pays a publisher each time an advertisement is clicked. A click is registered as a GET request to some URL. Various client identifiers attached to the GET request are checked to ensure clicks are coming from distinct clients. In click fraud, an attacker entices users to visit an attack page. This page generates a GET request to the advertisement URL, without any interaction from the user. Consequently, the advertiser pays for advertisements that were never seen.

Once again, it should be emphasized that little can be reliably concluded based on client authentication. Knowledge of the principal that sent a request is most often misleading information.

4. Related Work

All of the attacks described in this paper are presented in prior works. The Confused Deputy attack was originally presented by Norm Hardy in 1988 [8]. The term "Cross-Site Request Forgery" was coined in a blog post by Chris Shiflett in 2004. In 2000, a page on the Zope web site described a similar attack, naming it a "Client Side Trojan" [19]. Neither of these works made the connection to the Confused Deputy attack. In an August 2000 mailing list posting [17], Kragen Sitaker recognized the attack described by Zope as an instance of a Confused Deputy attack and described a variation of the attack using a web crawler instead of a web browser. The term "clickjacking" was coined for an OWASP Conference talk by Robert Hansen and Jeremiah Grossman [7]. Some of the rendering techniques used in the attack had been the subject of previous bug reports at Mozilla [14]. Shortly thereafter, clickjacking was recognized as a Confused Deputy attack in a web page by Tyler Close [3].

The realization that these attacks are not an artifact of any particular implementation but rather arise from a defect in the ACL model itself was implicit in the original Confused Deputy paper. This aspect of the problem was further elaborated upon in a paper by Miller et al in 2003 [11]. This discussion highlighted the loss of context that occurs in the ACL model due to the separate transmission of object identifiers and access rights. This paper also presented a refutation of the claimed equivalence of the ACL and capability models. This refutation pointed out the need for a global namespace for both principals and objects in the ACL model, where only local namespaces are used in the capability model. Though the paper separately discusses the Confused Deputy problem, the refutation of equivalence does not point out the differences in access decisions made by the two models, or how this semantic difference arises. In Dan Wallach's thesis [18], the Confused Deputy problem is briefly discussed in connection with stack introspection. This discussion does not include mention of the remaining Confused Deputy vulnerabilities in a stack introspection design discussed in this paper.

The core contribution of this paper is the description of the Confused Deputy problem in the terminology of the access matrix. This description clarifies how the ACL and capability models produce contradictory access decisions, thus providing a more exact characterization of the Confused Deputy problem. This precision enables a better description of the caused problems and the possible remedies, as well as a better understanding of how contemporary systems, like the Web, suffer from the problem. Better understanding of the inability of the ACL model to correctly control access in multi-party scenarios may help prevent the continued recurrence of these attacks.

5. Conclusion

In messaging scenarios involving more than two principals, the ACL model fails to retain enough information to enable correct access decisions. These errors in the model are manifested in all systems that use the ACL model for access control in multi-party scenarios. For example, these errors are the underlying flaw exploited by both the CSRF and clickjacking attacks on the Web. The capability model does not have these logic errors and can effectively control access in multi-party scenarios. Some systems, such as the Web, can be converted from the ACL model to the capability model without change to their infrastructure and with relatively minor changes to applications.

Acknowledgment

Thanks to Norm Hardy, David-Sarah Hopwood, Alan Karp, Chip Morningstar, Fred Spiessens and Marc Stiegler for providing valuable feedback on early drafts of this paper. Though the received feedback was instrumental in clarifying the presented arguments, any remaining confusion, or error, is the responsibility of the author.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis and A. Keromytis. The Role of Trust Management in Distributed Systems Security. Chapter in Secure Internet Programming: Security Issues for Mobile and Distributed Objects, (Vitek and Jensen, eds.) Springer-Verlag. 1999.

[2] Hao Chen, David Wagner and Drew Dean. Setuid Demystified. Proceedings of the 11th USENIX Security Symposium. 2002.

[3] Tyler Close. clickjacking: The Confused Deputy rides again! http://waterken.sourceforge.net/clickjacking/ . October 2008.

[4] Tyler Close. web-key: Mashing with Permission. IEEE W2SP 2008: Web 2.0 Security and Privacy. May 2008.

[5] D. F. Ferraiolo and D. R. Kuhn. Role Based Access Control. 15th National Computer Security Conference: 554-563. October 1992.

[6] Li Gong. Java 2 Platform Security Architecture. http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html . 2002.

[7] Robert Hansen and Jeremiah Grossman. Clickjacking. http://ha.ckers.org/blog/20080915/clickjacking/ . September 2008.

[8] Norm Hardy. The Confused Deputy (or why capabilities might have been invented). ACM SIGOPS Operating Systems Review, Volume 22, Issue 4, pages 36-38. October 1988.

[9] Butler Lampson. Protection. Proc. 5th Princeton Conf. on Information Sciences and Systems, p437, Princeton. 1971.

[10] Butler Lampson, Martin Abadi, Michael Burrows and Edward Wobber. Authentication in distributed systems: theory and practice. ACM Transactions on Computer Systems (TOCS), Volume 10, Issue 4. November 1992.

[11] Mark S. Miller, Ka-Ping Yee, and Jonathan S. Shapiro. Capability Myths Demolished. Technical Report Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University. Mar 2003.

[12] Mark S. Miller and Jonathan S. Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. 8th Asian Computing Science Conference (ASIAN03). 2003.

[13] Mark S. Miller, James E. Donnelley and Alan H. Karp. Delegating Responsibility in Digital Systems: Horton's "Who Done It?". 2nd USENIX Workshop on Hot Topic in Security (HotSec'07). 2007.

[14] Jesse Ruderman. iframe content background defaults to transparent. https://bugzilla.mozilla.org/show_bug.cgi?id=154957, mozilla.org. June 2002.

[15] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278.1308, September 1975.

[16] Chris Shiflett. Security Corner: Cross-Site Request Forgeries. http://shiflett.org/articles/cross-site-request-forgeries. Dec 2004.

[17] Kragen Sitaker. thoughts on capability security on the Web. http://lists.canonical.org/pipermail/kragen-tol/2000-August/000619.html , Kragen thinking out loud. August 2000.

[18] Dan S. Wallach. A New Approach to Mobile Code Security. PhD Thesis, Princeton University. January 1999

[19] Zope. Client side trojan issue. http://www.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan . 2000.