A General Framework for Categorizing Vulnerabilities Regarding Their Impact on Security Policy

Hamid Reza Shahriari\textsuperscript{a,*} Rasool Jalili\textsuperscript{a} Matt Bishop\textsuperscript{b}

\textsuperscript{a}Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
\textsuperscript{b}Department of Computer Science, University of California at Davis, CA, U.S.A

Abstract

This paper proposes formal definitions of the terms “vulnerability”, “exploit”, and “attack” in computer security. It suggests a general framework for categorizing vulnerabilities based on their impact on a security policy. As a case study, we develop and demonstrate useful categorization schemes from this framework. A wide range of vulnerabilities are analyzed to demonstrate the usefulness of the categorization schemes. Then countermeasures of vulnerability are developed for each category. We show that our framework is primitive and non-ambiguous. Moreover, the classes do not overlap, provided that the access rights observed in the system are independent.

Key words: Vulnerability, Vulnerability categorization, Vulnerability analysis, Attack, Exploit

1 Introduction

As networks grow, so does the diversity of attacks. Malicious users exploit vulnerabilities of computer systems to form attacks and violate system security. A proper categorization of attacks, based on their impact on the system, clarifies the attacks and the controls needed to mitigate them. Further, a control mitigating a specific category of attacks will mitigate all attacks in that

\* Corresponding author: Tel:+98(21)66164019, Fax:+98(21)66054097
\texttt{Email addresses: shahriari@ce.sharif.edu} (Hamid Reza Shahriari),\texttt{jalili@sharif.edu} (Rasool Jalili),\texttt{bishop@cs.ucdavis.edu} (Matt Bishop).

Preprint submitted to Elsevier Science 4 September 2006
category, making the control more effective than one targeting a specific attack. There is much published research on categorization of vulnerabilities from different points of view such as that of the attacked system component, the attacker’s goal, the attack’s result, the technique used in the attack, and the original weakness in the system caused the attack [1,2,3,4]. However, there have been few efforts to define ‘attack’ and ‘vulnerability’ formally and to categorize classes of attacks in a generic manner. Moreover, in previous works, the role of a security policy in the definition of ‘vulnerability’ has not been considered. Our goal is to study the impacts of vulnerabilities. In that sense, how or why they were introduced is irrelevant.

In this paper, we first propose a formal definition for ‘vulnerability’, ‘attack’, and ‘exploit’ based on the system security policy. Then, we propose a general framework for impact-based categorization of vulnerabilities. The categorization applies regardless of the nature of the security flaw or the attacker’s ultimate goal. Our categorization framework has many applications. It can be used to generate other impact-based categorization schemes. These impact-based categorization schemes are used in risk analysis, which is the process used to identify and evaluate risks and their potential effects. A risk analysis needs to specify the system vulnerabilities, their impacts, and the probability of their occurrence. Our framework can be easily deployed in any case and helps analysts to classify the system vulnerabilities and specify their effects on the system precisely. Moreover, it can be used in multi-phased attack detection, formal vulnerability analysis such as [5], and in log correlation for intrusion detection.

The remainder of the paper is organized as follows: Section 2 discusses the relationship of a security policy on vulnerabilities. Section 3 gives the basic definitions. Section 4 provides the general framework of the categorization. A case study and some examples in Section 5 show the applicability and flexibility of the framework. Section 6 concludes the paper with a discussion about the categorization framework.

2 Security Policy and Vulnerabilities

An often-overlooked aspect of vulnerability analysis is that a vulnerability is defined in terms of a security policy. When one speaks of a buffer overflow being a “vulnerability”, one implies that the buffer overflow can be exploited in a way that causes a violation of the specific site’s security policy.

As the basis of analyzing a system for vulnerabilities is the questioning of the assumptions that underlie the features and functionality of the system, let us ask whether a buffer overflow is always a security vulnerability. There are
several possible ways a policy can deal with that construct.

A security policy may state that “no buffer overflows may exist”. In this case, the existence of the buffer overflow violates the security policy, so it is clearly a vulnerability. However, few policies frame security issues at this low level, either explicitly or implicitly.

More typically, a security policy frames constraints upon the results of actions. A typical policy will state, for example, that no user may acquire privileges before they have been identified and authenticated as a user authorized to obtain those privileges. In this case, whether a buffer overflow is a security vulnerability becomes problematic.

Consider a stack-based buffer overflow. In this type of buffer overflow, the attacker overflows a buffer allocated on the stack with a machine-language program that will be executed, and then adds addresses so that when the function being executed returns, the return address used is that of the buffer and not that of the caller. As a result, the machine-language program is executed. For the purposes of our discussion, assume the machine-language program spawns a command interpreter.

The first program containing a stack-based buffer overflow is a web server. The attacker launches her attack and obtains a command interpreter on the targeted system. As the attacker has acquired privileges she did not have before (specifically, the ability to execute arbitrary commands on the system), the security policy has been violated. Here, the buffer overflow is a security vulnerability.

The second program is one that a student using the computer wrote. This test program copies a string from the command line to an internal buffer and prints the buffer. An attacker finds this program and executes it. The buffer overflow attack works, giving the attacker a command interpreter. But as the test program is unprivileged, so is the command interpreter, which therefore runs with the privileges of ... the attacker. As the attacker has not acquired any new privileges, this buffer overflow is not a security vulnerability.

As another example, consider an attacker who floods a gateway’s Internet connection, thereby cutting the company’s access to the Internet. If the company does not consider Internet availability a security issue, and can function perfectly well using its intranet (which is not affected by the attack), then the limited bandwidth of the connection to the gateway is not a security risk. But for a company like Amazon, whose business is conducted over the web and depends on users accessing its web server, the attack exploits the limited bandwidth, which—given the nature of the security policy requiring availability over the internet—makes that a security vulnerability.
The role of the security policy in defining vulnerabilities is critical, for without a known policy, one cannot say whether a programming or configuration or design error is in fact a security vulnerability. Some security policies are explicit; some are implicit; others mix both explicit statements and assumptions. But without a security policy, the question of whether something is a vulnerability is ill-posed, because the answer depends on the (non-existent) policy.

3 Basic Definitions

In this section we provide some basic definitions and use them to present our categorization framework.

**Definition 1** A computing system is a state machine $CS = (S, s_0, \tau, S_A)$, where $S$ is the set of all system states, $s_0 \in S$ is the initial state, $\tau$ is the set of state transitions, and $S_A \subseteq S$ is the set of authorized states. The set of unauthorized states is represented by $S_U = S \setminus S_A$.

The notion $s_i \xrightarrow{\tau_i} s_{i+1}$ means that the state transition $\tau_i \in \tau$ moves the system from state $s_i$ to state $s_{i+1}$. Also, the notion $s \xrightarrow{\ast} s'$ denotes that the system moves from state $s$ to state $s'$ using some state transitions. Then the state $s$ is also called a predecessor of state $s'$.

In the definition, the system protection states are partitioned into authorized and unauthorized states, such that $S_A \cap S_U = \emptyset$ and $S_A \cup S_U = S$. We utilize this partitioning to define the security policy as follows:

**Definition 2** Given a computing system $CS = (S, s_0, \tau, S_A)$, a security policy $P$ is a partition of the set of all states $S$ into authorized and unauthorized states, i.e. $P = (S_A, S_U)$, such that $S_A \cap S_U = \emptyset$ and $S_A \cup S_U = S$.

**Example** A site rule states that "users shall have those privileges for which they have authorization according to the site management manual". Each system state is fed into an oracle that does the following: "Return YES if, given the state, all users have exactly those permissions stated in the site management manual. Otherwise, return NO." The set of states for which the oracle returns YES is the set of authorized states $S_A$. The set of states for which the oracle returns NO is the set of unauthorized states $S_U$. As the oracle returns exactly one of YES or NO for each state, $S_A \cap S_U = \emptyset$. As the oracle returns an answer for every state that it is given as input, $S_A \cup S_U = S$. Hence this site rule is a security policy.

Bishop and Bailey defined a vulnerable state as "an authorized state from
which an unauthorized state can be reached using authorized state transitions" [6]. Accordingly, they define vulnerability as characterization of a vulnerable state which distinguishes it from non-vulnerable states. The definition implies that both states and transitions should be classified as authorized or unauthorized. We believe that, if the system is in an authorized state, unauthorized transitions from that state are disallowed. Accordingly, we define a vulnerable state as follows:

**Definition 3** Let \( CS = (S, s_0, \tau, S_A) \) be a computing system. The state \( s \in S_A \) is vulnerable iff there is a sequence of state transitions \( T = \langle t_1, t_2, \cdots, t_n \rangle \) where \( t_i \in \tau \ (1 \leq i \leq n) \) and states \( s_i \in S \ (1 \leq i \leq n) \) where state \( s_n \in S_U \), such that \( s \vdash t_1 s_1 \vdash t_2 \cdots \vdash t_n s_n \). \( s_n \) is called the compromised state, and the sequence of state transitions \( T \) is called attack.

**Example** A system at the site with the previous security policy runs a server with administrative privileges that accepts unvalidated input and stores it in a buffer on the stack. The server is vulnerable to a buffer overflow attack. Helen does not have access to administrator privileges. When the server is running, Helen sends the server input designed to produce a command interpreter as described in section 2. Her input overwrites the return address on the stack, causing the server to execute the input, thereby producing a command interpreter with administrator privileges that Helen can use. All states of the system in which the privileged server are running are vulnerable states. The state in which Helen has access to the command interpreter is a compromised state, because Helen has unauthorized privileges. The states in which she does not have access are not compromised states because Helen does not have administrator privileges. The transitions abstracting Helen’s uploading the input and the server’s response are the attack.

Regarding the above definition of vulnerable state, if there is a reachable vulnerable state in the system, all the states in the path beginning from the initial state to this vulnerable state are also vulnerable. The following theorem shows this:

**Theorem 1** Let \( s_j \) be a vulnerable state, and \( s_i \in S_A \) be a predecessor of state \( s_j \), i.e. \( s_i \vdash^* s_j \). Then state \( s_i \) is vulnerable.

**Proof.** The state \( s_j \) is vulnerable, so there is a state \( s' \in S_U \), such that \( s_j \vdash^* s' \). Also \( s_i \vdash^* s_j \), then we have \( s_i \vdash^* s' \), and it implies that \( s_i \) is vulnerable, which can be lead to unauthorized state \( s' \). □

 Accordingly, we distinguish the last vulnerable state in the attack that immediately precedes an unauthorized state.

1 Unless some user has unauthorized privileges, which is tangential to our example.
Definition 4 Let $CS = (S, S_0, \tau, S_A)$ be a computing system. State $s \in S_A$ is strictly vulnerable iff there are $t \in \tau$ and $s' \in S_U$ such that $s \vdash_t s'$. The state transition $t$ is called an exploit.

Example Return to the previous example. The state just before Helen has access to the command interpreter is the strictly vulnerable state. The transition giving Helen access to the command interpreter is the exploit. Assuming the usual level of abstraction for vulnerability analysis, the exploit is the execution of the command interpreter spawning code, the attack is the uploading of the code and its execution, and the strictly vulnerable state is the state in which the code has been uploaded but not yet executed (that is, the input routine has not returned, so although the return address on the stack has been changed, it has not been popped).

Based on the definition, an exploit is the last state transition that moves the system form a authorized state to unauthorized state (i.e. the compromised state).

4 The Categorization Framework

Exploiting vulnerabilities typically causes an alteration of access rights. An access control model represents the protection state. Vulnerabilities enable some subjects to acquire unauthorized access rights violating the system security policy. In this section, we categorize vulnerable states based on the obtained or denied access rights in the compromised state. This categorization does not depend on the security flaw nature or the attacker’s ultimate goal.

The set of unauthorized states may contain states in which a subject possesses a right to an object that violates the security policy, or where information can flow in a manner that violates the security policy. In the latter case, we may model this as a right that allows the information to flow. This right may be transient, existing only long enough for the information to flow in such a way that the policy is violated. Hence, we focus on exploiting vulnerabilities as a problem of altering rights.

The classification of intrusion types presented by Lindqvist and Johnson in [2] is based on the intrusion results, including the violation of confidentiality, integrity, and availability. Here, we formally define the general categories concentrating on the system security policy. These general categories provide a framework that can be used in any case.

In ordinary attacks, a malicious principal tends to acquire a special right
over an object. Therefore, a state will be unauthorized because the subject
(representing the attacker) has an extra right over an object. In some other
attacks, generally refereed as Denial of Service (DoS) attacks, the attacker
goal is denying access to a special object. Thus, in DoS attacks, the state is
unauthorized because an authorized subject does not have access to an object.

In any case, based on the access right acquired or deleted, we can distinguish
the vulnerable state via the involved right. Accordingly we categorize vulnera-
bilities based on the unauthorized existence, or lack of, the access rights in the
compromised state. Let $\rho$ be a generic right. Then the two generic categories
of vulnerabilities are as follows:

**Definition 5 ($\rho^+$-vulnerable)** For a computing system $CS = (S, s_0, \tau, S_A)$,
let $\rho$ be a generic right in the system. Then a state $s \in S_A$ is strictly $\rho^+$-
vulnerable iff there exist a state $s' \in S_U$ and a state transition $\omega \in \tau$ such that
$s \vdash_\omega s'$ and $\omega$ gives a right $\rho$ to a subject $s$.

**Definition 6 ($\rho^-$-vulnerable)** For a computing system $CS = (S, s_0, \tau, S_A)$,
let $\rho$ be a generic right in the system. Then a state $s \in S_A$ is strictly $\rho^-$-
vulnerable iff there exist a state $s' \in S_U$ and a state transition $\omega \in \tau$ such that
$s \vdash_\omega s'$ and $\omega$ revokes a right $\rho$ from a subject in $s$.

By defining the rights in any system, we can base the categories of vulnerabil-
ities on their impact. Accordingly, if there are $n$ types of rights in a protection
model, we can have $2^n$ categories of vulnerabilities. In the next section we
provide useful examples from real systems.

## 5 Case Study

We apply our framework to a protection model as an example. Read ($r$), write
($w$), and execute ($x$) are well-known as inert access rights over data in most
access control systems. In the Take-Grant model [7], the take and grant rights
are control rights. So in this particular case study, we consider $r$, $w$, and $x$
rights as the inert rights and $t$ and $g$ as the control rights.

Using definitions 5 and 6 for $\rho \in \{r, w, x, t, g\}$, we obtain ten categories of vul-
nerabilities. The informal definitions of the resulted categories are as follows:

- $r^+$: Disclosing. A vulnerability that may lead to disclosing of confidential
  information to unauthorized users. This is a violation of confidentiality.
- $w^+$: Altering. A vulnerability that may lead to an unauthorized write access
  on an object. It is also called modification. Integrity counters this vulnera-
  bility.
Table 1
Impact-based categorization of vulnerabilities using set of rights $R = \{r, w, x, t, g\}$.

<table>
<thead>
<tr>
<th>Access Right ($\rho$)</th>
<th>Unauthorized acquiring of the right</th>
<th>Unauthorized denial of acquiring of the right</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ($r$)</td>
<td>$r^+!$:Disclosing</td>
<td>$r^-!$:Concealing</td>
</tr>
<tr>
<td>write ($w$)</td>
<td>$w^+!$:Altering</td>
<td>$w^-!$:Locking</td>
</tr>
<tr>
<td>execute ($x$)</td>
<td>$x^+!$:Exposing</td>
<td>$x^-!$:Denying Service</td>
</tr>
<tr>
<td>take ($t$)</td>
<td>$t^+!$:Usurping</td>
<td>$t^-!$:Blocking</td>
</tr>
<tr>
<td>grant ($g$)</td>
<td>$g^+!$:Leaking</td>
<td>$g^-!$:Denying Delegation</td>
</tr>
</tbody>
</table>

- $x^+\!$: Exposing. A vulnerability that may make a service available to an unauthorized principal. This vulnerability leads to violation of exclusivity, which is part of confidentiality [2,8].
- $t^+\!$: Usurping. This vulnerability may cause a subject to get unauthorized control of a system or a part of a system. For example, when a Trojan horse is installed on a system successfully, then the attacker gets the control of the system in an unauthorized manner. Integrity and confidentiality counter this vulnerability.
- $g^+\!$: Leaking. A vulnerability that may cause an unauthorized subject to be delegated rights belonging to another subject.
- $r^-\!$: Concealing. A vulnerability that may prevent an authorized subject from reading an object. Availability counters this vulnerability.
- $w^-\!$: Locking. A vulnerability that may prevent an authorized subject from writing to an object. Availability counters this vulnerability.
- $x^-\!$: Denying Service. A vulnerability that may lead that may prevent an authorized subject from executing a program or accessing a service that, according to the security policy, he should be able to access. For example, a user cannot connect to a server, due to the attacker launching a flooding attack. Availability counters this vulnerability.
- $t^-\!$: Blocking. A vulnerability that may prevent an authorized subject from acquiring an authorized right. For example, when a user authorized to acquire systems privileges is blocked doing so.
- $g^-\!$: Denying Delegation. This vulnerability exists when a subject grants its authority to another subject, designating that subject as its delegate. A malicious subject may without authority cause the delegation to be revoked. For example, he/she may distribute a revocation list containing the certificate issued by the grantor. Integrity and availability counter this vulnerability.

Table 1 summarizes the categorization for $\rho \in R = \{r, w, x, t, g\}$ in a two-dimensional view.
5.1 Vulnerability Examples

In order to show applicability and usefulness of this categorization, we present some examples.

- **Plaintext Password in FTP Sessions**
  
  The FTP protocol transmits a plaintext password to authenticate its client users. A malicious attacker may sniff the transiting traffic and acquire the password. This is a *Disclosing* vulnerability.

- **Internet Explorer 6.0 HTA Vulnerability CVE-2006-1388**
  
  This vulnerability in Microsoft Internet Explorer 6.0 allows remote attackers to execute HTA (HTML Application) files. This is an *Exposing* vulnerability as the attacker acquires unauthorized execution right.

- **Using a FAT32 Filesystem in Microsoft Windows**
  
  A Microsoft Windows system may mount a FAT32-formatted file system, which provides no protection features. A user may access another user’s files on the same file system. Assuming the security policy denies users access to the other user’s files, the system may have one or more of the Disclosing, Altering, and Exposing vulnerabilities. Based on the disallowed user access type (read, write, or execute), the system moves to a compromised state in which it has an unauthorized read, write, or execute right.

- **Weak Password**
  
  Consider an operating system environment that uses the username-password authentication method. The security policy implies that no one is authorized to access a different user’s account. Let U be a user account protected with a weak password (i.e. it can be guessed easily, using a dictionary or brute-force attack). A malicious user Eve guesses U’s password and logs in to the account. This means that Eve takes control of U, which violates the system security policy. Eve acquires t (take) rights over U. Accordingly, the current system state is unauthorized and the previous state has the *Usurping* ($t^+$) vulnerability.

- **Buffer Overflow**
  
  Buffer overflow vulnerabilities are perhaps the most exploited vulnerabilities by network attacks [9]. As an example, CVE-2000-0218 is a buffer overflow vulnerability in the Linux mount and umount programs allowing local users to gain root privileges via a long relative pathname [10]. By exploiting the vulnerability, a malicious user can force the vulnerable process to execute arbitrary code. This is modeled by acquiring a take right over the process by the malicious user. The system state is now in an unauthorized state, assuming the security policy denies users the right to take the control of critical system processes. Thus, this buffer overflow is considered a *Usurping* ($t^+$) vulnerability.

- **Cross-Site Scripting**
  
  Cross-site scripting vulnerabilities have become increasingly common in
web applications. The essence of cross-site scripting (referred as CSS or XSS in the literature) is that an intruder causes a legitimate web server to send a page to a victim’s browser. This page contains a malicious script or HTML codes of the intruder’s choice. The malicious script is executed with the privileges of a legitimate script originating from the legitimate web server. Consequently, the attacker may access the client’s private data such as cookies, data files, or may access restricted web sites on behalf of the victim [11,12]. As an example, the vulnerabilities CVE-2002-0075 and CVE-2002-0682 are of this type [10]. Accordingly, CSS is categorized as a Disclosing (r+) vulnerability.

• **.rhosts Vulnerability**

Sometimes a user trusts another user and allows that other user to access resources as if she were the first user. One of the best examples of this is the .rhosts facility in UNIX systems. The .rhosts vulnerability occurs when a user trusts another user on a host or on the network. The first user adds the name and remote host of the trusted user in a file named .rhosts and located in the user’s home directory. These trusted users assume all the access rights of the user who trusts them. If the security policy denies such granting, then this is a Leaking (g+) vulnerability.

• **Flooding Attack**

A news agency has a web site on which it publishes the news. An attacker floods the web site. The attacker traffic absorbs so much bandwidth that users cannot access the web site to read the news. This is a Concealing (r−) vulnerability in the hosting network.

• **Log Clogging**

Attackers usually try to hide their actions to prevent detection of attack process. Suppose there is a log server that all machines in a network send their logs to. An attacker floods it by sending many SYN packets. Accordingly, other servers could not write their logs to the log server. This vulnerability is categorized as a Locking (w−) vulnerability.

• **Computer Viruses Overloading Resources**

Some viruses make many copy of themselves in the infected machine and consume all of some resources such as memory and CPU cycles. Then a legitimate user cannot execute any program. Not restricting the amount of resources that a program can use is a vulnerability in the class Denying Service (x−). Another example for the UNIX operating systems would be a program that runs fork() forever. If the number of fork calls is not limited, the process table may fill up and other users will be unable to execute any program.

• **Blocking Administrator Access**

In the UNIX operating system or the Windows XP system, we can consider the root or Administrator user as an entity that has take rights over all other subjects. In this example, the administrator is blocked from acquiring others’ rights, so in effect the administrator does not have take rights. This is categorized as a Blocking (t−) vulnerability.
One of the benefits of this categorization of primitives is the ability to develop countermeasures based on the primitives, and then apply them to specific vulnerabilities. The abstract countermeasures apply to all vulnerabilities in that class; the specifics of the countermeasures apply to each vulnerability.

Consider the class of “disclosing” information ($r^+$). A standard abstraction is to apply the access control matrix to block disclosure by denying the read right (or its equivalent) to the resource being protected. Let the security policy state that subject $s$ cannot read object $o$. Consider the mechanisms that may enforce this.

Two basic mechanisms exist: block access through some mechanism such as a capability list (C-List), or allow the subject to access the data in an unintelligible form, for example by enciphering the data. Which one is used depends upon the particular constraints of the environment.

For example, the three disclosing vulnerabilities in the previous section were the “Plaintext Password in FTP Sessions” vulnerability, the “Using a FAT32 Filesystem in Microsoft Windows” vulnerability, and the “Cross-site Scripting” vulnerability. Consider how a C-List might work for these three vulnerabilities. The FTP vulnerability requires access to the network. If a C-List could prevent such access, the vulnerability would not exist. The FAT32 vulnerability exists because of a lack of access controls, so the use of a C-List would solve that problem. Finally, if the C-List does not allow execution of scripts, or the sending of data to any third-party web site, then cross-site scripting fails.

The suitability of using a C-List depends on the environment and on the policies of the sites involved. For example, for the FTP vulnerability, a C-List is not likely to be effective unless the network in question is accessible only from systems which offer C-Lists that will control access to the network. In practice, few networks have these controls. So, enciphering the traffic would be more effective. In contrast, a system can control access to files in a file system, so adding C-Lists to Microsoft Windows would be effective. So would the application of appropriate cryptographic controls to create a cryptographic file system. An encipherment mechanism shared by the trusted site and the client would also solve cross-site scripting, provided all traffic and/or cookies sent to the trusted web site were enciphered appropriately; then the untrusted page could not read the critical values.

This demonstrates another advantage of our scheme; by classifying vulnerabilities and countermeasures using these primitives, defenders can see what countermeasures could be used against what vulnerabilities, and investigate
applications of countermeasures in new ways.

6 Discussion and Conclusion

In this paper, we proposed a general framework for categorizing and formally defining vulnerabilities and their exploits based on their impact on security policy. As it is a general framework, there may be different interpretations of the categorization in different systems based on the rights in the system. One common set of rights, which we have considered here, is \( r, w, x, t, g \), and this set has been explored as a case study in section 5.

The impact on the security policy is the most important criteria for classifying vulnerabilities [3]. However, the authors know of no previous work categorizing vulnerabilities in this manner. Our categorization was defined formally and so vulnerabilities can be categorized without ambiguity. Nevertheless, a vulnerable state may have multiple vulnerabilities. The main justification for this is the existence of multiple simultaneous vulnerabilities.

One of advantages of our categorization is that it is primitive. This means that each category has exactly one property: having or not having a specific right determines the vulnerability class. Thus, the categorization process is straightforward.

Another important advantage of our categorization is the classes are orthogonal. The basis of categorization is having an unauthorized right or not having an authorized right. Thus, if the rights are independent, the categories will be orthogonal. However, in some cases a right may include another right implicitly or explicitly. For example, if \( w \) (write) includes \( r \) (read) for some system, then the \( w^+ \) vulnerability implies the \( r^+ \) vulnerability, and the \( r^- \) vulnerability implies the \( w^- \) vulnerability.

One of the benefits of this categorization of primitives is the ability to develop countermeasures based on the primitives, and then apply them to specific vulnerabilities. The abstract countermeasures apply to all vulnerabilities in that class; the specifics of the countermeasures apply to each vulnerability.

It should be noted that one vulnerability may lead to another vulnerability. For example, in the FTP password vulnerability, the password is disclosed to a subject, so it is a Disclosing \( (r^+) \) vulnerability. Having the password, the subject may usurp control of the FTP account protected with the password, and then acquire the rights that account has over files. Depending on the rights that can be acquired, it could be a Disclosing or Altering vulnerability. Generally, this can be used to detect attacks composed of multiple sequential
exploits.

References


