Intrusion Ripple Analysis in Distributed Information Systems

Stephen S. Yau and Jun Zhu
Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287, U.S.A
Email: yau@asu.edu

Abstract: Security is a very important aspect of distributed computing systems, especially in distributed information environments involving wide-area networks, such as internets. In this paper, how a security breach, such as intrusion, propagates through a distributed or networked information system is addressed. In particular, the issue of how an intrusion, once occurred, propagates through a distributed system is addressed. Our approach has strong resemblance to the ripple effect analysis, which had been extensively studied in the area of software maintenance.

Keywords: Security, intrusion, ripple effect analysis, distributed or networked information systems.

1. Introduction

The advent of distributed processing and computer networks, like Internet, broadband networks, etc, has greatly increased our ability to process information globally. But at the same time, the distribution of sensitive data has also increased security risks. The computer systems in the distributed environment are vulnerable to both intrusion by outsiders and abuse by insiders. Security aspect of a distributed system is becoming more and more important as the computer security incidents are on the rise. Reid [1] discussed various problems that may happen within an open distributed system.

Security is an attribute of a system that includes confidentiality, integrity, non-denial of service, accountability, and authentication [2]. Confidentiality requires that the data in a computer system and transmitted information can only be revealed to individuals who are authorized to have access to the data. The protection of data from exposure includes not only the content of data itself, but also the existence of data. Integrity requires that the information in a computer system, as well as the transmitted information not be modified by unauthorized parties. Non-denial of service ensures that authorized users are not unduly denied access or use of any network access for which they are allowed. Some patterns of network attacks are often accomplished by blocking information flow entirely. Accountability is often referred to as non-repudiation which requires neither the sender nor the receiver of a message can deny the transmission. Authentication, which at the simplest form is password protection, is an essential component of a secure system. It prevents unauthorized users from accessing the systems’ sensitive data components. However, the authentication gateways can sometimes be compromised either by external penetrators or internal misfeasors. An intrusion may cause loss of confidentiality, loss of integrity, denial of services, or unauthorized use of system resources. In such cases, “intrusion detection” serves as the second line of defense against unauthorized access.

Intrusion detection has been studied extensively in [3,4], and some automated tools are available [5,6,7]. In general, an intrusion detection system (IDS) alerts a system security officer (SSO) when it detects an attack, and the SSO can take preventive action to eliminate the intruder and reduce the damages to the system.

Our approach to this problem is to analyze the degree to which the effect of an intrusion can permeate from the semantic/syntactical point of view when an intrusion occurs. Normally, in the information system of an organization, the components are related to each other. This relation can be described as a chain [8]. When an intruder breaks the authentication perimeter of the system, the sensitive data may have been revealed or corrupted. In this paper, the questions of which parts of the large (potentially enormous) distributed system
should be considered “under attack” due to this intrusion, and which parts can still be treated safe from a security standpoint are addressed. After these questions are resolved, then the SSO can focus on cleaning up the tainted components, and does not need to investigate other parts of the distributed system. The benefits are obvious in terms of reduced cost and effort. We will also address how a security breach, such as intrusion, propagates through a distributed or networked system.

Our approach has strong resemblance to the ripple effect analysis in software maintenance, by which changes to one program area have tendencies to be felt in other program areas [9]. Maintenance changes tend to spawn faults in other program areas because a programmer may fail to check all affected areas for consistency. The ripple effect analysis begins with an initial change in one area of the program, and identifies potentially affected areas in other modules of the program. Then, each potentially impacted area should be checked to see if it is spurious side effect or not. If so, appropriate changes should be made to maintain the consistency. Each of these changes should be handled in the same way as the initial changes. This process will continue until there are no secondary errors introduced in the program.

2. The intrusion propagation model

Bell and LaPadula [10] defined a data confidentiality model which includes security labels. The model specifies rule-based controls for reading necessary to preserve data confidentiality. This model also specifies the rule-based controls for writing to ensure that data is not copied to a container where confidentiality cannot be guaranteed. To analyze the propagation of intrusion, the data and control flows of the distributed system should be considered. In our approach, we use the access matrix and the dependency graph to investigate the propagation of the intrusion.

The access matrix is a configuration of a protection system, expressed as a tuple \((S, O, P)\), where \(S\) is the set of current subjects, \(O\) is the set of current objects, \(S \subseteq O\), and \(P\) is an access matrix, with a row for every subjects in \(S\) and a column for every object in \(O\). \(P[S, O]\) is a subset of \(R\), the generic rights. \(P[S, O]\) gives the rights to object \(o_i\) possessed by subject \(s_i\) and it could be \(w\) (write) or/and \(r\) (read).

In an access matrix, each subject represents a process, a program or a user, and each object represents data, a file, a program or a variable in a module, etc. So each subject is associated with a bunch of objects and each object is also associated with some subjects. If a subject wants to access the content of an object, it must have an ‘r’ entry in the corresponding position of the access matrix. Otherwise, the object is confidential to the subject.

For each row in the access matrix, all objects associated with the subject should have dependency relations with each other. Suppose \(s_i\) has access to the set of objects \(o_1, o_2, \ldots, o_n\) and the dependency graph for \(s_i\) is shown in Fig. 1, where the arrowhead from \(o_i\) to \(o_j\) represents that the second one is computed from the first one. It also implies that subject \(s_i\) has at least read access to \(o_j\) and write access to \(o_i\). Each object associated with \(s_i\) may be associated with some other subjects. Then, Fig. 1 can be expanded by including the dependency graphs of \(s_i\) and the subjects which have any common objects. For example, Fig. 2 is an example of the dependency graph of objects to which three subjects \(s_j, s_2, s_3\) have accesses. When an intrusion occurs in this case, depending on which part of the system is revealed to the intruder, the intrusion will permeate through the system differently. As a simple case, suppose the intrusion happens, and object \(o_1\) is revealed to the intruder. Based on the propagation paths shown in the dependency graph, we should consider objects \(o_2, o_3, o_4\) as tainted objects. \(o_7\) and \(o_8\) are objects that every subject has access to. They are global variables.
3. Definitions and Rules

In this section, we will define the terms and rules used in our intrusion ripple analysis.

**Definition 1**: For each subject $s_i$, the associated object set is $O_{s_i} = \{ o_j \mid o_j \text{ has an entry in the access matrix for } s_i \}$.

**Definition 2**: For each object $o_i$, the associated subject set is $S_{o_i} = \{ s_j \mid s_j \text{ has an entry in the access matrix for } o_i \}$.

**Definition 3**: For each object set associated with $S_o$, the source capable set is $C_{so}$, where $C_{so} \subseteq O_{so}$. $C_{so}$ is the set of objects in $O_{so}$ which cause potential error within and flowing from $O_{so}$. The potential propagator set $P_{so} \subseteq O_{so}$ is the set of all usage in $O_{so}$ which can cause objects in the source capable set to flow from $O_{so}$. The flow mapping is $Cso \leftarrow f(P_{so})$.

**Rule 1**: If we have a flow mapping: $o_2 \leftarrow f(o_1)$, where $o_1$ and $o_2$ are all single objects, then (a) if $o_1$ is a tainted object, $o_2$ will also be considered tainted and (b) if $o_2$ is a tainted object, $o_1$ will also be considered tainted. These two cases are represented graphically in Fig. 3.

**Rule 2**: For a flow mapping: $o_3 \leftarrow f(o_1, o_2)$, where $o_1$, $o_2$, and $o_3$ are all single objects, then (a) if $o_1$ is revealed and $o_2$ is not, then $o_3$ should be considered safe, (b) if both $o_1$ and $o_2$ are revealed, then $o_3$ should be considered tainted, (c) if $o_1$ and $o_2$ (or $o_2$) are revealed, then $o_2$ or $o_1$ may or may not be tainted, depending upon how $o_3$ is generated. In order to include the possibility that $o_2$ is tainted, we assume that $o_2$ is tainted in Rule 2.c. Fig. 4 below illustrates this rule.

4. Characteristics of intrusion propagation

In this section, we will observe several characteristics of intrusion propagation.

**Explicit vs. Implicit**

Our approach to the intrusion propagation analysis is from the semantic/syntactic point of view. When we construct the dependency graph of the system and consider the intrusion propagating through that system, we can see that objects are not necessarily have computational relationship with each other. One object may be explicitly computed from another object (explicit), but may also be implied from the other object (implicit). For example, if the intruder somehow knows the zip code of an employee’s address, he could certainly know the area the employee lives. During the ripple effect analysis, we should consider both explicit and implicit relations between objects.

**Global, semi-global and local**

The global objects are the objects that are associated with all subjects. The semi-global objects are those which are accessible to more than one subject. A local object is only associated with one subject. Revealing global objects in the distributed system has bigger impact than local objects. Since they are accessible to all modules who use them.

**Inclusive vs. Exclusive**

As specified in Section 3, the intrusion to the system may have inclusive or exclusive characteristic. The exclusive characteristic specifies that one object is computed exclusively from another object, and hence the revelation of the latter will cause the revelation of the former. On the other hand, there may have several objects served as input for computing the object. Then, if not all of the input objects is known by the intruder, the output object is not necessarily revealed.

**Forward and Backward**

The intrusion usually propagates forward in the direction of the dependence and data flow. But, it may also go backward to the object which serves as input. For example, $B=A/2$. The intrusion certainly will go forward from $A$ to $B$ with respect of flow mapping $B \leftarrow f(A)$ if $A$ is revealed, but it can also go backward from $B$ to $A$ if $B$ is revealed. Rule 2(c) is also a case.
The data confidentiality requires that both data in a computer and the data transmitted in the communication channel should not be revealed to the unauthorized user. This means that both Persistent and Non-persistent objects should be considered. A Persistent object could be a file, or anything stored in disks. A Non-Persistent object could be a variable which will only exist in the memory of computer systems. Instead of breaking into the system by cracking the authentication perimeter and then getting access to the persistent data of the system, the intruder could also steal the sensitive non-persistent information by monitoring the network traffic.

5. Intrusion ripple analysis

To analyze intrusion propagation, we need first to identify at which point the intrusion happened and thus what sensitive data has been revealed as a result of the intrusion.

We assume that the initial tainted data has been found, such as by means of analyzing the audit trails. As we specified before, when the tainted sensitive data moves through the network system, the damage will increase because some other data may be computed from this data.

Intrusion ripple analysis is an iterative process until we reach the point that all tainted data components in the system are identified. One important aspect of the ripple effect analysis is the granularity. It could have two levels: inter-modular or intra-modular. If more accurate result is desired, the intra-modular level should be used and the cost of the analysis is going to be high. On the other hand, if less accurate result is sufficient, the inter-modular level analysis should be used and the cost is going to be low. These two levels can be used together in some cases.

Our approach to the intrusion propagation analysis consists of the following basic steps:

1) Decide the granularity level of the analysis. Following are some factors which affect the choice of the granularity level. We may choose either one of them or both. In most of case, both of the levels should be used.

- More detailed analysis is needed when we need to have better understanding.
- The more detailed analysis should be done to the portion of the system where the intrusion happened and those modules closely related to the intruded portion because when the intruder break into the system, he may have best understanding to the system just around the point he intruded the system.

2). The amount of cost and effort we want to put on the analysis.

2. Perform the logical dependency analysis on the components of the system, including the analysis the access rights of the subjects to the objects and construction of the access matrix.

2.1). Construct the access matrix. And for each subject, identify the dependency relations between objects associated with it so the source capable set, potential propagator set and the flow mapping could be constructed.

2.2). Construct and analyze the dependency graph, identify if the tainted objects are local, semi-global and global objects. This can be done by analyze the scope of the objects.

2.3). Identify any implicit dependency relations among objects not in the dependency graph. To accomplish this step, we need have good understanding of the system and the domain knowledge regarding any possible dependency relations among objects not explicitly shown in the dependency graph.

3). Apply the ripple effect algorithm (rules) iteratively to identify tainted components. Stop the tracing until no new secondary propagation source.

- Starting from the initial tainted set, applying the rules to the dependency graph constructed from the previous step to identify the additional tainted objects.
- Add the tainted objects to the initial tainted object set and repeat the previous step until the initial tainted object set no longer change.

6. An example

To illustrate the concept of the intrusion ripple analysis, let us consider the following example. Suppose we have a networked environment that is abstracted to a transport connection [12] in which calling and called modules can successfully transmit messages and parameters to implement a subroutine call. We assume that there are three procedures residing in three different nodes shown in Fig. 5, which is part of a distributed database system. It implements the functionality of
accessing a remote database system according to the query language provided. It will also verify the access right of the caller by checking the password. There are three modules which are named queryinterface, queryRLDB, and accessTuples respectively. The parameters of these three procedures are all passed by values. Fig. 6 shows some parts of three procedures.

Step 1). We decide to choose intra-module granularity to do the analysis because we need to have good understanding of this system.

Following Step 2.1), we construct the access matrix of the program as follows: three modules are treated as three subjects \( s_1, s_2 \) and \( s_3 \) for procedure queryinterface, queryRLDB, and accessTuples, respectively. The access matrix in this example is given in Fig. 7. From the access matrix in Fig. 7, we can see that there is no global variable, and \( s_2 \) (procedure queryRLDB) have access to all objects. By Definitions 1 to 3, we have \( O_{s_1} = \{ \text{query_input, userid, passwd, res} \} \), \( C_{s_1} = \{ \} \), \( P_{s_1} = \{ \} \), and there are no dependency relations among those variables. Hence, no flow mapping function can be constructed.

\[
O_{s_2} = \{ \text{query_input, userid, passwd, res, access_level, rev_query, t, tname, pwd} \}, \quad C_{s_2} = \{ \text{access_level, rev_query, tname, t, res} \}, \quad P_{s_2} = \{ \text{userid, passwd, query_input, access_level, rev_query, pwd, tname, t} \},
\]

and the flow mapping functions are \( \text{access_level} \leftarrow \text{f(userid, passwd)} \), \( \text{rev_query} \leftarrow \text{f(query_input, access_level)} \), \( \text{tname} \leftarrow \text{f(rev_query)} \), \( \text{t} \leftarrow \text{f(pwd, tname)} \), \( \text{res} \leftarrow \text{f(t, rev_query)} \).

\[
O_{s_3} = \{ \text{t, pwd, tname} \}, \quad C_{s_3} = \{ \text{t} \}, \quad P_{s_3} = \{ \text{pwd, tname} \}, \quad \text{and the flow mapping functions are} \quad \text{t} \leftarrow \text{f(pwd, tname)}. \]

queryinterface()
{
    ...
    queryRLDB(query_input, userid, passwd);
    res = get_result();
    ...
}

queryRLDB(NL query_input, char *userid, char *passwd)
{
    int access_level;
    Tuple *t;
    Char *pwd; // password for access the third node.
    ...
    if ( (access_level=check_privilege(userid, passwd)) !=0 )
    {
        ...
        rev_query=revise_query(query_input, access_level);
        tname=get_table_name(rev_query);
        accessTuples(tname, pwd);
        t=recvieve_tuples();
        ...;
        process_query(rev_query, t);
        send_result();
        ...
    }
    else exit();
    ...
}

accessTuples(char *tname, char *pwd)
{
    BOOL has_right;
    ...
    open_password_table();
    has_right = authenticate(pwd);
    if (has_right) then
        send_tuple(tname);
    else deny_access();
    ...
}

Step 2.2). According to the dependency relations generated in Step 1), the dependency graph can be constructed as shown in Fig. 8.

Step 2.3). We cannot identify any implicit dependency relations among objects in the dependency graph.

Now, we can introduce some intrusion to the system to see what will happen. In our example we assume that the intruder has access to variables query_input, userid, and passwd in the first procedure. Then, we know that

<table>
<thead>
<tr>
<th>query_input</th>
<th>userid</th>
<th>passwd</th>
<th>access_level</th>
<th>rev_query</th>
<th>res</th>
<th>tname</th>
<th>t</th>
<th>pwd</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
<td>( r )</td>
</tr>
</tbody>
</table>

Fig. 7 The access matrix of the example.
the initial tainted set is \{query_input, userid, passwd\}. According to Step 3), the analysis can be done as following:

1). The initial tainted set is \(T = \{query\_input, userid, passwd\}\).

2). For object set associated with \(s_1\), since there is no flow mapping function, we skip it and continue to analyze \(s_2\). By applying Rule 2.b to \(access\_level \leftarrow f(userid, passwd)\), we know that \(access\_level\) is tainted. Tainted set now should be \(\{query\_input, userid, passwd, access\_level\}\). Then, we can apply Rule 2.b to \(rev\_query \leftarrow f(query\_input, access\_level)\) to identify \(rev\_query\) as another tainted variable. Now, the tainted object set is \(\{query\_input, userid, passwd, access\_level, rev\_query\}\). After that, by applying Rule 1.a to \(tname \leftarrow f(rev\_query)\), we find that \(tname\) is tainted and add it to the tainted \(T\).

3). At this point, we can no longer find other tainted objects because the flow mapping functions related to \(res, t, and pwd\) are \(t \leftarrow f(pwd, tname)\), and \(res \leftarrow ft(rev\_query)\). Those situations can be eliminated by applying Rule 2.a to them. So, the tainted set \(T\) would be \(\{query\_input, userid, passwd, access\_level, rev\_query, tname\}\). All other variables including the result \(res\) are still safe.

7. Conclusion

In this paper, we have discussed the framework of a technique for analyzing the intrusion ripple effect. The concept here is by utilizing the ripple effect analysis technique in software maintenance area, identify all objects which are revealed to the intruder. Since we have only presented the analysis from the confidentiality point of view, additional work needs to be done for integrity, non-denial of service, accountability etc.

References: