Client-Side Deception Using Architectural Degradation

Dennis Heimbigner
Computer Science Department
University of Colorado
Boulder, CO 80309-0430, USA

Abstract

The problem of thwarting insider misuse of software systems is addressed by modifying the behavior of software systems to provide deceptive responses to detected or suspected insider attacks in order to mitigate their effects. The approach is based on the controlled architectural degradation of the software by using dynamic reconfiguration. Specific reconfigurations are provided for a software system so that at least the following range of deceptions can be implemented when misuse is detected or suspected: 1) Misdirection – generate inaccurate or imprecise information in response to requests by the misuser; 2) Operational degradation – remove selected pieces of functionality so that the misuser is deprived of those functions; 3) Self-destruction – modify the software so that it appears to be working although it has had all important information and capabilities destroyed, ideally without immediate knowledge by the misuser; 4) Restoration – undo deceptive responses.

Keywords: deception, insider threat, intrusion detection.

1. Introduction

Detecting and controlling insider misuse of software systems is increasingly recognized as an important problem, especially for military systems [5][6][7]. Detecting misuse of software may be viewed as a special case of the intrusion detection problem [10]. Detection per se is out of the scope of this effort, and it is assumed that some mechanism exists to detect misuse or at least alert a security administrator about suspicious behavior [1].

Once detected, it is important to provide some sort of response that has the ability to control any malicious actions taken by the insider. One possible (but foolish) policy for controlling misuse would be to cause a software system to self-destruct whenever any misuse alarm is generated. In the face of even a moderate number of false positives, this policy would of course lead to chaos.

The goal we propose is to deceive the client – the insider – about the capabilities of the software and the contents of datasets used by that client so that it becomes harder to perform malicious acts or to steal important data. The solution proposed here is to deliberately degrade the operation of the software and data used by the client.

The approach is to reconfigure the software and data to achieve degraded operation. This involves applying a controlled sequence of responses (changes to the system). The general class of changes involves deception, which provides an increasing amount of misinformation to the misuser. This approach allows for better response to false alarms while retaining the ability to disarm (and later rearm) the software by degrees. In this approach, false alarms are met initially with measured responses that provide both time and information necessary to verify the correctness of the alarm. As the seriousness of the misuse is ascertained, more powerful deception responses can be brought to bear.

Our alternative also provides the capability to back-out of enacted responses: in effect, to rearm the software. If it is quickly determined that no misuse is occurring, then any responses taken to that point should be reversible so that the full and efficient functionality of the software is restored. The term "restoration" is used to refer to this ability to return to operational effectiveness.

This approach is based on the controlled reconfiguration of the software and data. The term "reconfiguration" is defined broadly to include changes in parameters, the addition, modification, or removal of functional components of the software system, and modification, or removal of datasets. Specific and pre-defined reconfigurations are

provided for a system so that at least the following range of deceptive responses can be implemented when misuse is suspected:

- Operational degradation – the ability to reconfigure the software to remove selected pieces of functionality so that the misuser is deprived of those functions.
- Misdirection – the ability to modify the software and data to generate inaccurate or imprecise information in response to requests by the misuser.
- Self-destruction – the ability to reconfigure the software and data so that it appears to be correct but has in fact had all important information and capabilities destroyed, ideally without immediate knowledge by the misuser.

In addition to the above, it is also expected that reconfiguration can be used to provide enhanced monitoring; this is the ability to reconfigure the software to generate extra logging information to aid in detecting true misuse.

Finally, the ability to reconfigure the software to undo degradation response is provided. This is a unique capability made possible by our use of dynamic reconfiguration, and is important in the event of false alarms or the external neutralization of the misuse threat.

2. Architectural Degradation

To pursue the previously defined classes of responses, it was necessary to narrow the focus to provide a working prototype that could be made to exhibit these responses and that was itself commonly used.

As part of this narrowing, it was recognized that the problem of insider response naturally split into two related parts, each of which could be independently pursued. These two parts were (1) document-based systems, and (2) function-based systems.

By document-based, it is meant that for many enterprises, including many military enterprises, the creation and management of documents is the primary activity. Most importantly, only a limited set of software programs is utilized in this situation: editors, word processors, spreadsheets, and especially World Wide Web (Web for short) browsers.

By contrast, function-based systems utilize a larger variety of software programs, each of which has some significant computational functionality. Examples might be fire-control systems and logistics systems. In each case, the program is not generic in the sense that a word processor is generic.

Since the set of programs was more constrained, it was decided to focus on document systems: that is, the unique problems of degrading document-based systems, and more specifically, on degrading web access. Degrading function-based systems was to be pursued later in the project.

With the focus narrowed to document-based systems, the next steps were to define attack scenarios, choose some degradable software, and establish the initial prototype.

3. Degradable Web Access Prototype

Figure 1 illustrates the traditional web access system. The client side and the server side usually represent two different sites on a network. The server side is composed of a web server (such as Apache) and a number of backend data sources such as collections of web pages or databases that provide the data exported by the web server.

The primary interface on the client side is the web browser such as Internet Explorer (IE) or Firefox. Often, a special proxy is placed between the browser and the servers to provide additional functionality such as caching or access control. In addition, it is usual for a firewall to mediate all access between the client side and the larger network.

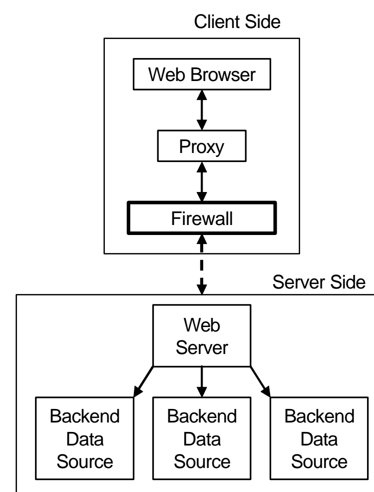


Figure 1. Typical Web Access System

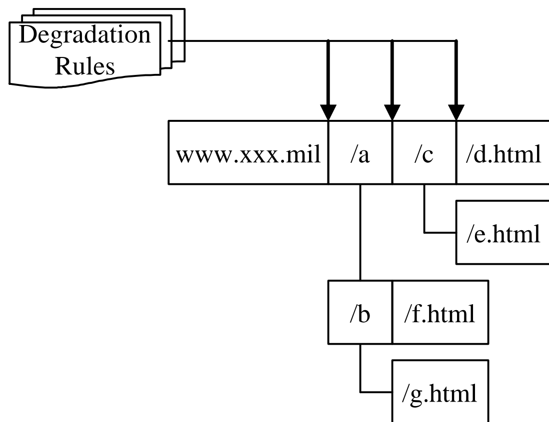
This may be less common in situations where the network is relatively homogeneous and trusted.

4. Degrading a Web Based System

For web based and other document systems, the notion of operational degradation is rather restricted. One can of course cause a browser or word processor to refuse to view, modify, or save a document. While these are important limits on functionality, they do not really get at the essence of the idea of degrading the insider's capabilities.

Internally, documents often have significant structure based on, for example, the Extensible Markup Language (XML), but this pertains to only an individual document and not to sets of related documents. Web sites also have informal structure, but this structure is rarely made explicit and formal. So, one of our goals is to define some form of regular, formal structure for organizing a set of web accessible documents. We wish this structure to provide some equivalent to an architecture so that we can define degradation concepts over it.

We must also address the issues of where degradation is to occur and where it is initiated. Our goal is to degrade access for specific clients. This can be accomplished at various points. In Figure 1, for example, degradation can occur at any (or all) of the components on the client side: browser, proxy, or firewall. Alternatively, degradation can also occur on the server side if the client side reliably notifies the server about the identity of the client. Server-side degradation has some drawbacks because then it can only apply to the information obtained by the insider from that server. Placing it at the client side allows us to control the degradation of all the data for a given insider.



URL Space: www.xxx.mil
www.xxx.mil/a/c/d.html
www.xxx.mil/a/c/e.html
www.xxx.mil/b/f.html
www.xxx.mil/b/g.html

Figure 2. URL-Based Document Set Architecture

Ideally, degradation should occur in the browser because it has the most information and the best interface to the insider. However, modern browsers are extremely complex pieces of software and so we compromise and insert our degradation into the proxy. This is a temporary choice, and as we gain a handle on the structure of browsers, we anticipate moving more of the degradation capabilities into the browser.

Initiating degradation must occur at two levels. Most immediately, we assume that our browser or proxy has an associated set of rules that control when local degradations are initiated. Beyond that, we must assume our browser/proxy also sends out messages indicating events that are occurring at the browser. These events can be received by secondary monitoring processes (including security administrators) that in turn can invoke additional degradation activities.

5. A Web Site Architecture

Our essential insight is that the URL structure of web pages provides a natural "architecture" for one or more web sites. We will use the term "web data" to refer to the collection of web pages accessible through some web server.

Figure 2 illustrates our notion of URL-based architecture for web data. The idea is that we take all the URLs of all the web pages and form them into a tree rooted, in this case, at the base URL "www.xxx.mil" (the "http://" is elided). The figure shows a tree constructed from the URL paths at the bottom of the figure.

Each square node in the figure represents some element in the path. Degradation rules (Section 7) are associated with each possible subpath. Documents under the same node presumably are related. We will defer the formalities associated with constructing a tree for now and appeal to intuition about that process.

6. Degradation Proxy Capabilities

Our initial implementation involved modifying a web proxy rather than a web browser. We did this primarily because of the difficulties we experienced in understanding the internal structure of a browser of the complexity of Firefox. Our modified proxy supports the functional capabilities described in the following subsections.

6.1 URL Subtree Monitoring

The ability to monitor access to some subtree of documents is perhaps the most important capability we can provide. This is because it provides infrastructure necessary to most other capabilities. It is also essential in allowing security administrators insight into document access patterns.

When monitoring is established on a URL subtree, it generates events for every access into a specific URL prefix that represents the subtree. The specific items to be included are specified when the monitoring is invoked as the consequent of a rule firing. The event includes some subset of the following information: Complete URL path, Identity of the accessor, Date and time, and Action.

The last item – action – is a defined set of actions that the accessor might perform at any time on the data within the monitored URL subtree. If for example, the "file.save_as" action was enabled, then any software system that read the document and later tried to save it would generate this event. Obviously, some actions can only be enabled if monitoring is embedded into a relevant program such as the browser. For our initial attempt using a proxy, this set of actions is somewhat limited.

6.2 URL Subtree Suppression

The goal of the suppression operation is to make it effectively impossible for an accessor to retrieve items from within a specified URL subtree. Within the domain of web access, there are essentially four techniques, two of which are derived from the basic mechanisms provided through the HTTP protocol.

1. *404 Not Found*: We are all familiar with this response indicating a "broken link." It occurs when a reference to a URL does not lead to an existing web page. We can mark a subtree so those attempts to reference items in the subtree generate this kind of deceptive error, even though, of course, the web page is still there.
2. Timeout: Similar to the 404 message, this indicates that an attempt was made to contact a web server but that no response was obtained after some time-out period passed. The timeout usually is caused when the server itself is down, although it may also occur when the server is obtaining the data from some backend source (Figure 2) and that source is not responding. In our context, timeout becomes a mechanism for preventing access to selected subtrees.
3. *Crashing*: Occasionally, access to a page will cause the browser to crash. Simulating a crash can be an effective mechanism for preventing or delaying access to data.
4. *Invisibility*: Sometimes it may be desirable to completely hide the existence of some URL subtree. This requires scanning all web pages when accessed to remove or transform all instances of URLs referring to the invisible subtree.

6.3 URL Subtree Substitution

Substitution involves replacing one whole URL subtree with another subtree. Mechanically, this is relatively simple to do, but in practice, it is costly because it requires the construction of a plausible substitute set of documents to replace those in the original tree.

7. Degradation Rules for URL Trees

The preliminary sets of conditions and actions that we are implementing as part of our rule system are shown in Tables 1 and 2.

As is usual, rules consist of two parts: a condition and an action. Whenever the condition is met, the associated set of actions is executed. In our model, each rule is attached to one or more nodes in our URL tree. Whenever an access is made that refers to that URL, then the rules are checked.

There is one ambiguity that must be resolved. Suppose we have two URLs: *www.xxx.gov/a* and *www.xxx.gov/a/b*. Further suppose each of these URLs has some set of attached rules: When a reference to *www.xxx.gov/a/b/x.html* occurs, which rules should be checked: those associated with URL 1? Should it be those associated with URL 2, or both? We assume that rules can be marked as "exact," "prefix," or both. An exact rule is checked only when its URL is an exact match. A prefix rule

Table 1. Condition Operators

Condition	Semantics
Reference	If this subtree is referenced
Modified	If a document in this subtree is modified.
Scan(%)	If a subtree has had x% of its subtree scanned since the last scan-reset() action.
Execute(class)	Execute a specific method of the specified Java class. If the method returns true, then treat it as a satisfied condition.

is checked only when the URL is a prefix of the actual URL. A rule can have both designations, in which case it is invoked in either case. However, it is invoked only once no matter what, so if the rule needs to take into account if it is exact or prefix, then it is probably better to write two rules.

8. The Deceptive Web Proxy Architecture

We chose the Rabbit Web Proxy program [9] as the basis for building our degradation supporting web proxy. It is attractive both because of its relative simplicity, its support for the HTTP 1.1 protocol, and because it is implemented in Java; this latter point is important for portability across multiple computing platforms.

The Rabbit proxy must be augmented with a number of additional components in order to support the capabilities described above.

- *Event Notification*. The proxy acts as a Siena client (Section 9.2). This allows the proxy to generate event messages that can be received at any other connected client, and especially at the security administrator's console (Section 9.3).
- *Rule Interpreter***.** As URL references are made, they are matched against the rule sets. Matches require the interpretation of the rules to carry out the necessary degradation operations.
- *Rule Management.* We need to maintain a database of rules and URL paths for use by the interpreter. We also need to provide access to the local rule sets for the security administrator.

Table 2. Action Operators

Action	Semantics
Monitor	Generate an event and send it to Siena.
NotFound(path)	Suppress any occurrences of the specified path in this subtree by generating a "404 Not Found" error.
Timeout(path, sec)	Suppress any occurrences of the specified path in this subtree by generating a timeout error after sec seconds have passed.
Crash(path)	Cause Browser fails whenever the specified path is accessed.
Invisible(path)	Suppress any occurrences of the specified path in this subtree by making the path invisible.
Replace(path1,path2)	Substitute path2 for path1 in this subtree.
Scan-reset	Reset scan testing on this subtree
Execute(class)	Execute a specific method of the specified Java class.

9. Auxiliary Programs

In addition to the proxy, we need some auxiliary programs to simplify operation, maintenance, and management of the degradation proxy.

9.1 URL Tree Based Rule Editor

We use simple text files for our initial specification of URL subtree rules. But as soon as is feasible, we plan to construct an editor to support the definition of these rules.

9.2 Siena Event Notification System

Event distribution is carried out using the University of Colorado Siena publish/subscribe system [2]. In a publish/subscribe system, clients publish event (or notification) messages with highly structured content, and other clients make available

a filter (a kind of pattern) specifying a subscription: the content of events to be received at that client. Siena notification messages are structured as attribute-value pairs where attributes are simple names and the value is taken from a limited set of primitive types. A filter is a set of triples of the form (attribute, operator, value). A filter matches a notification if the value associated with each attribute in the notification satisfies all corresponding filter triples that have the same attribute name.

9.3 Security Administrator's Console

To make the deception functions accessible, we provide a program that allows a security administrator to see the events generated by the monitoring function in the degradation proxy. This console controls loading, activation, modification, and de-activation of rules and allows the security administrator to generally monitor the operation of the deceptive web proxy. Figure 3 shows the interface for the initial version of this tool.

10. Postponed Issues

We recognize that a number of issues cannot be handled by a proxy-based implementation. Rather, they must await the construction of a true browser-based implementation.

- *Forcing Use of the Proxy*: As long as we depend on using a proxy, we must have some mechanism for enforcing use of the proxy. The only way we know of to do this is to force specific firewall rules that only allow the proxy to access external URLs. Another possible solution is to require use of a Secure Sockets Layer (SSL) and allow only the proxy to access the web servers.
- *Restoration*: In the event of a false alarm where it is determined that an access pattern is benign, it will be necessary to undo the misinformation. Our basic approach is to modify the browser to raise a flag that some information in the history of accessed pages has changed. The changed information will, of course, represent the correct (as opposed to degraded) information and the user will be encouraged to view the new data.
- *Javascript*: Javascript is ubiquitous in web pages, and we must have some reasonable approach for handling it. The long-term solution is to modify the Javascript interpreter used in the browser to make it handle our degradation rules. This is not generally possible at the proxy, and so this will have to wait until we can modify the browser.

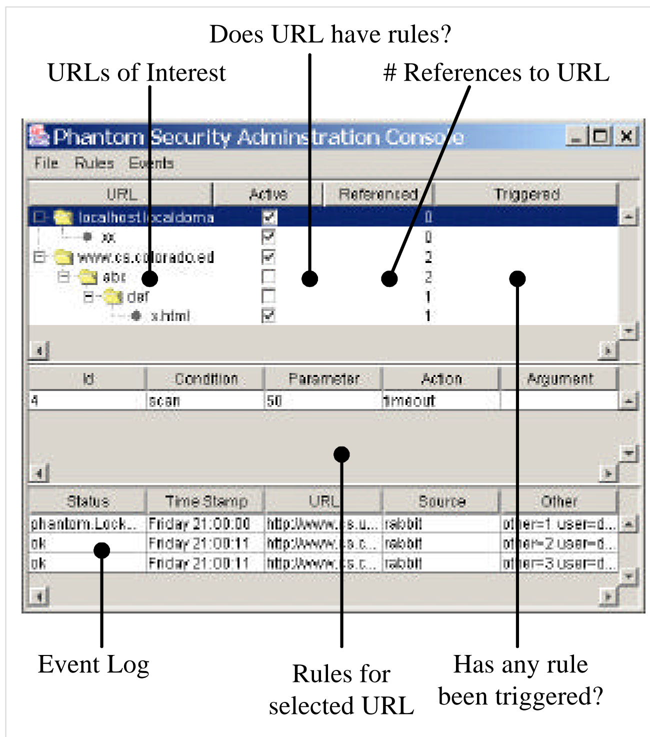


Figure 3. Security Administrator's Console

11. Related Work

The idea of using deception was first proposed by Fred Cohen [3][4]. That work to date has focused on server-side deception and on honeypots [11]. The idea is to provide a fake domain (a honeypot) with a set of plausible computers operating various services (e.g., FTP, HTTP). The goal is to make this honeypot accessible on the Internet and to allow external attackers to attack the honeypot. Any activity against the honeypot is assumed to be malicious because there is, by definition, no legitimate activity on it. By capturing information about accesses to the honeypot, it is possible to gain significant information about external attackers and their methods.

The client-side deception proposed differs from honeypots in that it is initiated by the system against a potential insider attacker. Thus it modifies a system for which the insider has legitimate access

and use to provide a modified environment with varying degrees of deception added to it.

The idea of using reconfiguration comes from our previous work with Willow [8][12] The Willow project used reconfiguration to provide a degree of intrusion tolerance for distributed system. It operated by reconfiguring the system either to repair it or to harden (and later un-harden) the system against some specific kinds of attacks.

12. Future Work

Our goal is to develop and generalize the range of possible deception that can be accommodated by our architecture degradation approach.

An interesting application may be battlefield security. As soldiers bring powerful, hand-held devices onto the battlefield, they become targets for enemy capture and subversion. Detecting such captures and deliberately feeding false information to the enemy would provide a useful tool.

13. Conclusion

The problem of thwarting insider misuse of software systems is addressed by providing responses to detected or suspected insider attacks in order to mitigate their effects. The approach is based on the controlled reconfiguration of the software to cause it to act in deceptive ways. We have developed the infrastructure for demonstrating such client-side deception using a web browser.

14. References

[1] Anderson, K., A. Carzaniga, D. Heimbigner, and A. Wolf, “Event-based Document Sensing for Insider Threats,” Computer Science Department Technical Report CU-CS-968-04, University of Colorado, February 2004.

[2] Carzaniga, A., D.S. Rosenblum, and A.L. Wolf, “Design and Evaluation of a Wide-Area Event Notification Service”, ACM TOCS 19(3):332-383 (August 2001).

[3] Cohen, F. “A Mathematical Structure of Simple Defensive Network Deceptions”. Fred Cohen and Associates Technical Report, 1999, (http://all.net/journal/deception/mathdeception/mathdeception.html).

[4] Cohen, F., D.Lambert, C. Preston, N. Berry, C. Stewart, and E. Thomas. “A Framework for Deception”. Fred Cohen and Associates Technical Report, July 13, 2001, (http://all.net/journal/deception/Framework/Framework.html).

[5] Dept. of Defense, “DoD Insider Threat Migration: Final Report of the Insider Threat Integrated Process Team,” April 24, 2000.

[6] Hayden, M., “The Insider Threat to U.S. Government Information Systems,” National Security Telecommunications and Information Systems Security Committee. July 1999.

[7] Jones, A.K. (Chair), “Cyber-Security and the Insider Threat to Classified Information,” Computer Science and Telecommunications Board, November 1-2, 2000.

[8] Knight, J., D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, and P. Devanbu, “The Willow Survivability Architecture,” Proc. of the Fourth Information Survivability Workshop, Vancouver, B.C, March 2001.

[9] Rabbit Web Proxy Home Page at Source Forge. (http://rabbit-proxy.sourceforge.net/)

[10] Snapp, S., J. Brentano, G. Dias, T. Goan, L. Heberlein, C. Ho, K. Levitt, B. Mukherjee, S. Smaha, T. Grance, D. Teal, and D. Mansur, “DIDS (Distributed Intrusion Detection System) - Motivation, architecture and an early prototype,” Proc. of the 14th National Computer Security Conference, Washington, D. C., Oct. 1991, 167 – 176

[11] Spitzner, L., *Honeypots: Tracking Hackers*, Addison-Wesley, 2002.

[12] Wolf, A., D. Heimbigner, J. Knight, P. Devanbu, M. Gertz, and A. Carzaniga. “Bend, Don't Break: Using Reconfiguration to Achieve Survivability”. Proc. 2000 Int’l Survivability Workshop. Boston, Massachusetts, October 2000.