State maintenance in Web applications

Jeni Li
July 24, 2002
GSEC v1.4, option 1

Abstract

As each transaction between Web client and Web server occurs in a stateless environment, state must somehow be passed from one transaction to the next in a Web application. State maintenance is one of the major vulnerabilities in Web applications and application environments today, whether commercial, open-source, or home-grown. Flawed state maintenance techniques can leave an application vulnerable to user impersonation, unauthorized data access, or outright systems compromise.

This paper will provide an overview of the four basic methods for keeping state in Web applications (in the URL, in hidden form fields, in cookies, and on the server side), discuss security considerations in using these methods, and offer some recommendations for keeping state securely.

1 Introduction

In a survey of 45 Web applications in production at client companies, @stake researchers found that 31% of e-commerce applications examined were vulnerable to session replay or hijacking -- the highest incidence of Web application security defects encountered in the study. @stake's Andrew Jaquith (2002) comments, "user session security remains the Achilles heel of most e-business applications."

State maintenance (also known as session maintenance, or session state) is a familiar issue for Web programmers. HTTP is a stateless protocol (Fielding et al, 1999). For our purpose, this means that each transaction between Web client and Web server occurs in a brand-new environment with no recollection of previous transactions. For any reasonably complex Web application, this means that state must somehow be passed from one transaction to the next.

This paper will provide an overview of the four basic methods for keeping state in Web applications. It will then discuss security considerations in using these methods and offer some recommendations for keeping state securely.

Some Web application development packages (both commercial and open source) provide state maintenance mechanisms to assist the developer. These mechanisms are simply black-box functions of the same basic state maintenance methods, and the application developer is well advised to examine such functions before relying on them for application security.

2 State maintenance methods

There are four basic ways to keep state in Web applications:

- pass it in the URL
- pass it in hidden form fields
- set a cookie
- keep it on the server side

Even in the case of server-side state maintenance, the session state is still maintained and reported by the client; the difference is that the application gives the client a session id and stores the rest of the state data on the server side, associated with the session id. These are also the only options available to application environments (e.g., ASP, php, ColdFusion) that offer the programmer a black-box approach such as a Session object. Really.

2.1 In the URL (path info or query string)

If you've used a popular search engine such as Yahoo or Google, you are familiar with state passed in the URL. The technique is simple: State information is tacked onto the URL after the file location, preceded by a slash (known as path info) or a question mark (known as the query string). An example URL containing state data is shown below:

```
http://somesite.com/somescript.cgi?var1=foo&var2=bar
```

In a CGI application, this information is available in environment variables (PATH_INFO and QUERY_STRING) created by the Web server. In ASP and similar development environments, the information is available as properties of a Request object or similar construct. Web applications can pass information in this way by redirecting the browser to the new URL (e.g., with a Location header) or through form submission using the GET method.

Passing state in the URL is handy for some applications. A particular session state can be bookmarked, or called from other sites. The technique will work with absolutely any browser (but it should be noted that use of the "Back" button can be a serious problem with this technique, as state will be lost by backtracking through the URL history).

On the other hand, state data in the URL can be easily detected, modified, and reposted... and therefore faked. It may also be exposed in the URL history on the client machine, and in the access logs of proxy servers and Web servers.

2.2 Hidden form fields

Passing state in hidden form fields should work with any browser that supports forms. State passed in hidden form fields will be treated as the query string if the form method is GET, or as standard input if the form method is POST. Using hidden form fields with the POST method generally looks prettier than passing state in the URL. An example HTML tag indicating a hidden form field is shown below.

```
<input type="hidden" name="var3" value="foobar">
```

Those form fields aren't totally hidden, of course; the fields are visible by viewing the HTML source. And the form can be copied, modified locally, and reposted. Web programmers may try to work around this issue by checking the Referer header (in CGI, the HTTP_REFERER environment variable) to verify that it contains the URL of the form they expect to call their application. This is a mistake (Stein, 1998). The Referer header is reported by the browser; it can therefore be forged using browser proxy tools such as WebProxy (Swiderski, 2002, in software references) and RFProxy (Rain Forest Puppy, 2001, in software references) -- or even by simply telnetting to the appropriate port on the Web server (usually port 80) and entering HTTP headers and commands by hand.

2.3 Cookies

Cookies are pieces of information saved in memory and/or a text file on the browsing machine as directed by the server. On replying to a request from the client, the Web application returns a Set-Cookie header containing session information. This instructs the client software to return a Cookie header containing the same session information to the domain and path specified by the Web application when making future requests.

Timeouts can be set on cookies, assuming the browser complies. If a timeout is set, the cookie is written to disk; this means the state information can persist across browser sessions on a given machine. If no timeout is set, the cookie remains in memory and becomes unavailable after the browser is closed.

The Set-Cookie header in common use is formatted as follows (Netscape, 1995):

```
Set-Cookie: NAME=VALUE; expires=Wdy, DD-Mon-YYYY
HH:MM:SS GMT; Path=PATH; Domain=DOMAIN_NAME; Secure
```

NAME and VALUE are the name and value, respectively, of the state information (e.g, a variable) as specified by the Web application. All other parameters are optional. The expires parameter sets a timeout as mentioned above, based on the time according to the client machine. The Domain and Path parameters specify request conditions (host names and file locations, respectively) under which the client software will send a Cookie header. The Secure parameter means that the client will send the Cookie header only when the connection is SSL-encrypted or otherwise deemed by the client to be secure.

Note: The specification given here is an ad-hoc standard developed by Netscape Corp. The proposed standard for cookies varies somewhat from this use; in particular, the fixed expiration time is replaced with an optional "Max-Age" parameter indicating the maximum age of the cookie in seconds. Details of this can be found in IETF RFC2965 (Kristol & Montulli, 2000).

Cookies are not immediately visible to the user; however, it is possible to view and/or change them. Cookies are not supported by all browsers, but are supported by most browsers in use today. However, many users are suspicious of them (due to privacy concerns over targeted marketing à la DoubleClick and similar applications) and will turn them off.

Cookies are commonly considered to be fairly secure and unchangeable; however, a modified browser or browser proxy such as WebProxy (Swiderski, 2002) or RFProxy (Rain Forest Puppy, 2001) would readily report any cookie desired to any domain desired.

2.4 Server-side state

Instead of storing the data on the client, a session ID is stored on the client using one of the previously described techniques. The data are stored on the server side (e.g., in a temporary file or database table) and associated with the session ID. Using server-side state maintenance, a session might still be hijacked, but the state data can't be faked without compromising the data source. Nor is server-side state data accessible in access logs or client history.

Keeping state data on the server side is generally considered the safest and most appropriate technique when handling information of a sensitive nature.

3 Security considerations

State-related threats to Web application security include:

- state forgery (falsification of state data on submission)
- session forgery (guessing and impersonating a valid session)
- session replay and hijacking (detecting and impersonating a valid session)
- malicious input (specially formatted input designed to compromise the application)
- "session residue" on public clients (leading to potential session replay and/or compromise of sensitive user information)

This section will discuss these threats and countermeasures against them.

3.1 State forgery

State variables reported by the client are subject to falsification. For example, a user may modify state data in the URL and resubmit the request, or post a locally saved copy of an HTML form with modified hidden form fields. As previously mentioned, the user can modify state data passed in cookies as well.

Many shopping cart applications (both free and commercial) have displayed classic state falsification vulnerabilities, allowing users to modify product prices and quantities before submitting order forms (ISS X-Force, 2000). In applications such as Mambo Site Server 3.0.x (Miro Pty., Ltd., 2001, in software references), which set a generic "authenticated" or "admin" variable, state falsification may be used to bypass user authentication and gain administrative privileges within the application (Palomo, 2001).

Countermeasures against state forgery include the following:

- Keep state data on the server side.
- Verify client-side state data with a hash.
- Avoid relying on global variables, or at least make sure they're explicit.

3.1.1 Keep state data on the server side

As mentioned above, in this case the browser maintains only a session id. The session id is related to relevant state data on the server side -- as a unique key in a database table, for example.

Some Web applications using server-side state maintenance have attempted to use the client IP address to identify a client’s session. This is a mistake. Given the use of proxy servers by major ISPs such as America Online and major corporations such as Motorola, client IP addresses are neither unique nor consistent.

Additional care must be taken when using server-side state maintenance with multiple, load-balanced servers, as in a Web farm. Depending on the Web farm's configuration, it may be possible for a single user session to begin on one server but continue on a different server. Therefore, state data should not be kept on local disk on each server, as this could result in a session losing state partway through a transaction. State data may instead be kept in a file or database accessible to all servers in the farm, or in some similar construct. This raises another issue: With multiple servers writing state data to the same source, race conditions may occur if file locking is not handled appropriately.

3.1.2 Verify client-side state data with a hash

Stein (1998) suggested a workaround for verifying state information that is stored on the client side: Before passing state data to the client, generate an MD5 hash of all critical state variables (e.g., product prices and quantities in the shopping cart example) plus a hash key known only to the application, and pass the hash as well. On receiving subsequent requests from the client, the application can regenerate the hash from the reported critical variables and the secret hash key. If the regenerated hash is different from the reported hash, then the variables have been tampered with and the application can handle that condition accordingly (e.g., reject the session as invalid and/or log the tampering attempt).

This is a reasonable workaround, especially if the secret hash key can be changed regularly (e.g., daily), but server-side state maintenance is still a more reliable safeguard against falsification of state data.

3.1.3 Avoid relying on global variables, or at least make sure they're explicitly set

The Mambo Site Server 3.0.x vulnerability (Palomo, 2001) may serve to illustrate this problem and the countermeasure. In vulnerable versions, a login page would check the login credentials entered by the user. If the user successfully authenticated as an application administrator, the login page would set a global variable indicating administrative privileges and pass control to a separate page. The second page did not check user credentials, but relied on the global variable for authorization. Unfortunately, the second page could be called directly with the global variable set in the URL, bypassing user authentication altogether.

The countermeasure in this case is straightforward: Don't do that. Any global variables used in the application should always be set explicitly by the application; however, it is better to avoid using them altogether (Rafail, 2001). Depending on the Web application environment, there may also be some configuration options to safeguard against this problem.

3.2 Session forgery

Session forgery is really state forgery using a session id. By passing a different session identifier to the Web application, the attacker hopes to assume the identity of a legitimate user and conduct application transactions as that user.

Applications are susceptible to session forgery when they rely on predictable session identifiers to validate users. A "predictable" session id may be derived from user information (e.g., account numbers) or an obvious id generation algorithm (e.g., numbers or strings that follow a predictable sequence). For example, a commercial fax server application included a Web client that generated session ids according to a set pattern. Knowing one session id, a user could guess other session ids and use those ids to access other users' faxes and documents (Torres, 2000).

Countermeasures against session forgery include the following:

- Generate random, complex session identifiers.
- Enforce session timeouts.

3.2.1 Generate random, complex session identifiers

Valid session ids should not be easily guessed or inferred from the id of an active session. Account numbers, userids, and sequential numbers are examples of Really Bad Session Ids. A good session id has nothing to do with the user or the state of the session, aside from being related on the server side; it is long, complex, and truly random.

"Truly random" deserves passing mention here. If session ids are generated using randomizing functions, but are predictable, then they are not truly random. Ensuring randomness is outside the scope of this paper; however, further information can be found in Eastlake et al (1994).

3.2.2 Enforce session timeouts

By restricting the time a session id is valid, the Web application can reduce an attacker's opportunity to forge a valid session. If a given transaction is expected to take ten minutes to complete, for example, session ids may be expired after ten (or perhaps fifteen) minutes. A measure of flexibility can be attained by tinkering with the time limit, or even renewing a session's validity with each legitimate hit.

Sessions should be timed out according to timestamps generated on the server side and saved with each session, not according to expiration times on the client (e.g., a cookie expiration timestamp). In this way, a session that has timed out cannot be "resuscitated" by forging a cookie header. Depending on the design of the application, some garbage collection may be necessary to purge expired sessions.

Why expire sessions based on the server's time and not the client's? Aside from the question of trusting the client to respect the time limit, time zone issues on the client side may render the application unusable. For example, this author's employer in Arizona deployed a Web application with a session timeout of ten minutes. In addition to timing out sessions on the server side, the application specified cookies with expiration dates in GMT (the time zone required by the Netscape cookie spec). The application rolled out without problems in February;

however, in April, many users complained that the site didn't work. After much head-scratching and many failed attempts to recreate the problem, it dawned on a programmer that the users with problems had computers set to the Pacific time zone -- which had just switched to Daylight Savings Time, which Arizona does not observe. The cookies were being expired before they were set.

3.3 Session replay and hijacking

Using sniffing or cross-site scripting, a malicious third party may obtain a legitimate user's session id and use that id to impersonate the user (Kolsek, 2000). A malicious third party may also take advantage of application design flaws such as race conditions in state table maintenance (Phuzzy L0gic, 2001).

Local users on a Web system may have additional opportunities to detect and forge valid session ids, depending on the architecture and configuration of applications and their environments. For example, the state maintenance functionality built into php 4.x writes session ids into the /tmp directory by default (Lorch, 2002). By browsing this directory, any local user could detect current session ids and use them in a session hijacking attempt. This may be of particular concern in multi-user commercial Web hosting environments.

Countermeasures against session replay and hijacking include the following:

- Use SSL.
- Enforce session timeouts.

3.3.1 Use SSL

To avoid session id detection through sniffing, use SSL to encrypt the session whether the transmitted information is sensitive or not (Jaquith, 2002).

As a corollary to this, if session ids are passed as cookies, the cookies should be specified using the "secure" parameter. If the "secure" parameter is omitted, the browser will send the same session id to the server via both encrypted and unencrypted sessions. This may allow cross-site scripting exploits to obtain a session id (Kolsek, 2000).

3.3.2 Enforce session timeouts

This countermeasure has been discussed above. Enforcing reasonable session timeouts on the server side can reduce the effectiveness of sniffing exploits involving, say, sending the user a malicious email message containing HTML that invokes an unencrypted connection to the target server when the message is opened or previewed (Kolsek, 2000).

3.4 Malicious input

Session ids and state data should be treated as any other user input. It cannot be assumed that the client will report session data as it was set by the application. A malicious user may attempt to compromise a Web application by modifying session data. This includes not only user input to a form, but any information reported by the client machine -- cookies, referring URLs, browser identification strings, et cetera. An application relying on any of this information may be susceptible to buffer overflows, SQL injection, or any of the usual malicious-input exploits.

Depending on the application and server configuration, SQL injection and other malicious input exploits may be used to bypass user authentication, access user data (including login credentials and account information), insert or modify existing user data, or even execute arbitrary stored procedures or system commands in a privileged context (Rain Forest Puppy, 1999; Rain Forest Puppy, 2001; SK, 2002; Yamazaki, 2002).

Countermeasures against malicious input include the following:

- Untaint and validate session ids and state data.
- Beware buffer overflow.

3.4.1 Untaint and validate session ids and state data

All user input and other information reported by the browser should be untainted and validated before it is used. There are two approaches to untainting input, which we may call positive and negative. The positive untainting approach specifies the format or structure of a well-formed input string and accepts nothing else. The negative untainting approach specifies invalid characters and looks for them in the input string, rejecting them (or the entire string) if any are found. The positive approach is preferred. In most cases, we may not know every type of evil input that could break an application, but we do know what types of input are good and expected (CERT Coordination Center, 1999).

Much has been written on this topic, and it is somewhat outside the scope of this paper; however, the topic is critical to the security of any Web application. For further reference on this topic, see Stein & Stewart (2002) and Rain Forest Puppy (1999).

3.4.2 Beware buffer overflow

Avoid static memory allocation and check content length. This may be an argument for using hidden form fields and the POST method, as the length of posted input is available in the CGI environment variable CONTENT_LENGTH. By checking this variable, the application can determine the length of the posted data before handling the input.

3.5 "Session residue" on public clients

In the case of a kiosk or public-access client (e.g., in a library, computer lab, or "Internet café"), information remaining on the computer after a session may be a vulnerability. After the user walks away from the public client, a new user may view cached pages, URL history, and cookies on the disk. In this way, the original user's personal information may be compromised, and the potential for session replay is introduced.

This can be especially problematic if valid, active session ids or authentication credentials remain on the client after the session is over. A commercial travel booking Web portal was recently found to store userids and passwords in cookies in clear text (Sutton, 2002). While a session might be encrypted and therefore safe from sniffing attacks, an attacker with physical access to the machine could obtain authentication credentials by simply viewing the cookie, a process that is as simple as opening a text file on the disk.

Countermeasures against "session residue" on public clients include the following:

- Enforce session timeouts.
- Encrypt any authentication credentials that must be stored on the client.
- Avoid GET method and passing session ids in the URL.
- If using cookies, use only per-session cookies.
- Use non-caching directives.

3.5.1 Enforce session timeouts

This countermeasure has been discussed above. Enforcing reasonable session timeouts on the server side can reduce the risk of a new user exploiting a previous user's session, simply by limiting the time period during which the session is valid.

3.5.2 Encrypt any authentication credentials that must be stored on the client

The better approach, of course, is never to store authentication credentials on the client machine (e.g., in a cookie). However, user convenience may trump security considerations in some cases, as when users would like to have a "remember my id and password" function in an application. In this case, such credentials should be encrypted rather than stored in plain text (Sutton, 2002).

3.5.3 Avoid GET method and passing session ids in the URL

Session information passed using the GET method appears in the URL. Session information in the URL may be visible in the browser's URL history. If it is likely that an application will be accessed from a public client, the POST method or cookies should be used instead.

3.5.4 If using cookies, use only per-session cookies

Specifying a cookie without an expiration time will result in the browser keeping the cookie in memory rather than writing it to disk. As soon as the browser process is closed on the client machine, the cookie will disappear. Again, user convenience may trump this countermeasure in the type of case described above. Where possible, however, specifying a per-session cookie rather than a persistent cookie is a useful countermeasure.

3.5.5 Use non-caching directives

Intermediate forms containing session data in hidden form fields may be visible in cache files generated by the browser (Sutton, 2002) or, incidentally, maintained by a proxy server. The application may specify additional HTTP headers that instruct the browser and any proxy servers not to cache such forms. Not all clients will support these headers, but putting them in doesn't hurt. The syntax of appropriate headers is given below:

```
Cache-Control: no-store
Pragma: no-cache
Expires: Wdy, DD-Mon-YYYY HH:MM:SS GMT
```

Use all three. The Cache-Control directive shown will instruct HTTP/1.1-compliant browsers and proxy servers not to save the response to non-volatile media (Fielding et al, 1999). The Pragma: no-cache and Expires directives (compatible with HTTP/1.0-compliant clients) do not explicitly direct clients not to store responses (Ibid.), but clients may refrain from storing a non-cached/expired response. In the Expires directive, the date can be the current time (according to the time on the Web server) or some time in the past.

4 Conclusion: The state of state maintenance on the Web

Several years ago, a few projects were undertaken by W3C working groups addressing state maintenance. Strictly from the application security perspective, these projects seem to have little to offer beyond state maintenance techniques currently in use.

The most relevant project resulted in a working draft on session identification (Hallam-Baker & Connolly, 1996). This would have added an HTTP request header, Session-id, with a session identifier generated by the browser. The session id would then be used as a unique identifier in a server-side state maintenance scheme of the application developer's design (or perhaps in a manner built into the Web server or application environment, as is currently the case with ASP, php, ColdFusion, and other environments with built-in session support). From the application security perspective, this approach is no different from server-side state maintenance using a cookie for the session id -- except that the client would now determine what that session id looks like. In particular, session ids could still be forged or contain malicious input.

The HTTP/1.1 specification (Fielding et al, 1999) did not incorporate the Session-id request header and is silent on state maintenance in general, except to indicate that the From request header (intended to contain the user's email address) should not be used for that purpose. As the protocol involves simple text communications between the client and the server, it is hard to imagine an extension or modification to the protocol that could prevent a malicious client from lying about its state.

Given the stateless nature of the protocol and the likelihood that making the protocol "stateful" would still involve some sort of negotiation between client and server with the client reporting state, it would seem that the basic state maintenance issues aren't going to go away anytime soon. Happily, much of the risk can be minimized using plain old good programming practice, which is available right now.

5 References

5.1 Online references

CERT Coordination Center. "How to remove meta-characters from user-supplied data in CGI scripts." February 1999. URL: http://www.cert.org/tech_tips/cgi_metacharacters.html (21 Jul. 2002).

Eastlake, Donald; Stephen Crocker; Jeffrey Schiller. "Randomness recommendations for security." December 1994. URL: http://www.ietf.org/rfc/rfc1750.txt (21 Jul. 2002).

Fielding, Roy; Jim Gettys; Jeffrey Mogul; Henrik Frystyk Nielsen; Larry Masinter; Paul Leach; Tim Berners-Lee. "Hypertext Transfer Protocol -- HTTP/1.1." June 1999. URL: http://www.ietf.org/rfc/rfc2616.txt (21 Jul. 2002).

Hallam-Baker, Phillip; Dan Connolly. "Session identification URI." 1996. Rev. 960221. URL: http://www.w3.org/TR/WD-session-id/ (21 Jul. 2002).

ISS X-Force. "Form tampering vulnerabilities in several Web-based shopping cart applications." June 2000. URL: http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?id=advise42 (21 Jul. 2002).

Jaquith, Andrew. "The importance of application security." March 2002. URL: http://www.atstake.com/research/reports/acrobat/atstake_application_security.pdf (21 Jul. 2002).

Kolsek, Mitja. "Remote retrieval of IIS session cookies from Web browsers." July 2000. URL: http://www.acros.si/aspr/ASPR-2000-07-22-1-PUB.txt (21 Jul. 2002).

Kristol, Dave; Lou Montulli. "HTTP state management mechanism." October 2000. URL: http://www.ietf.org/rfc/rfc2965.txt (21 Jul. 2002).

Lorch, Daniel. "PHP 4.x session spoofing." Bugtraq Mailing List. January 2002. URL: http://online.securityfocus.com/archive/1/250196 (21 Jul. 2002).

Netscape. "Client side state - HTTP cookies." 1995. URL: http://wp.netscape.com/newsref/std/cookie_spec.html (21 Jul. 2002).

Palomo, Ismael. "Serious security hole in Mambo Site Server version 3.0.X." Bugtraq Mailing List. July 2001. URL: http://online.securityfocus.com/archive/1/199414 (21 Jul. 2002).

Phuzzy L0gic. "NMRC advisory - NetDynamics session ID is reusable." November 2001. URL: http://www.nmrc.org/advise/netdynamics1.txt (21 Jul. 2002).

Rafail, Jason. "CERT/CC vulnerability note VU#874803: php variables passed from the browser are stored in global context." rev. 32. October 2001. URL: http://www.kb.cert.org/vuls/id/847803 (21 Jul. 2002).

Rain Forest Puppy. "Perl CGI problems." Phrack 9(55). September 1999. URL: http://www.wiretrip.net/rfp/p/doc.asp/i2/d6.htm (21 Jul. 2002).

Rain Forest Puppy. "RFP2101: RFPlutonium to fuel your PHP-Nuke." February 2001. URL: http://www.wiretrip.net/rfp/p/doc.asp/i2/d60.htm (21 Jul. 2002).

SK. "SQL injection walkthrough." May 2002. URL: http://www.securiteam.com/securityreviews/5DP0N1P76E.html (21 Jul. 2002).

Stein, Lincoln. "Referer refresher." WebTechniques 3(9). September 1998. URL: http://www.webtechniques.com/archives/1998/09/webm/ (21 Jul. 2002).

Stein, Lincoln; John Stewart. "The World Wide Web security FAQ." v. 3.1.2. February 2002. URL: http://www.w3.org/Security/Faq/ (21 Jul. 2002).

Sutton, Michael. "Datalex BookIt! Consumer password vulnerabilities." Bugtraq Mailing List. June 2002. URL: http://online.securityfocus.com/archive/1/276215 (21 Jul. 2002).

Torres, Efrain. "RightFax Web Client 5.2: Hijack user's sessions." Bugtraq Mailing List. January 2000. URL: http://online.securityfocus.com/archive/1/44245 (21 Jul. 2002).

Yamazaki, Keigo. "Webmin/Usermin session ID spoofing vulnerability." SNS Advisory #53. May 2002. URL: http://www.lac.co.jp/security/english/snsadv_e/53_e.html (21 Jul. 2002).

5.2 Software references

Miro Pty., Ltd. Mambo Site Server. March 2001. v. 3.0.x php source. URL: http://www.mamboserver.com/ (21 Jul. 2002).

Rain Forest Puppy. RFProxy. March 2001. v. 0.pr perl source. URL: http://www.wiretrip.net/rfp/talks/cansecwest-2001/ (21 Jul. 2002).

Swiderski, Frank. WebProxy. April 2002. v. 1.0 binaries for Windows, Linux, and Solaris. URL: http://www.atstake.com/research/tools/index.html#vulnerability_scanning (21 Jul. 2002).