USP SECURE ENTRY SERVER®

Documentation series

UNITED SECURITY PROVIDERS

Secure Login Service

Operators Guide

Version 5.19.0.4

United Security Providers AG
www.united-security-providers.ch
info@united-security-providers.ch

Headquarter Stauffacherstrasse 65/15 CH-3014 Bern Tel. +41 31 959 02 02
Baslerpark Mürtschenstrasse 27 CH-8048 Zürich Tel. +41 44 496 61 11

UNITED SECURITY PROVIDERS

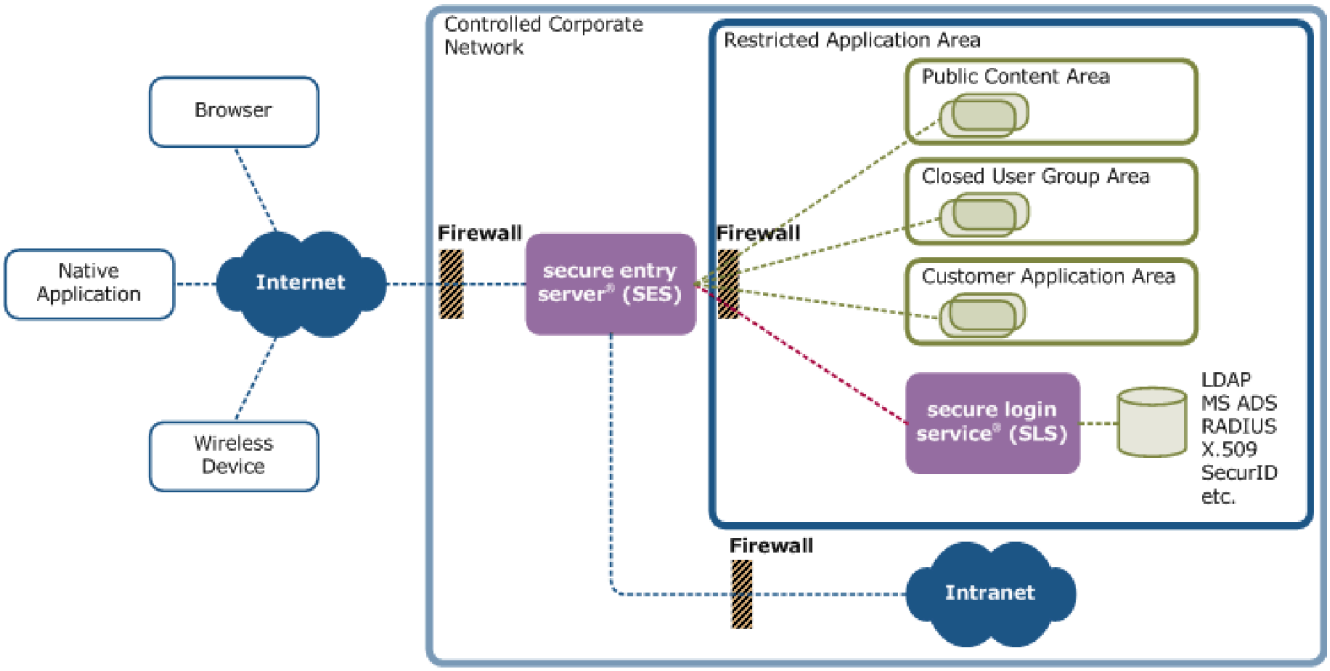
Contents

Chapter 1

SES Overview

1.1 Introduction

It is crucial to have a correct understanding of the SES infrastructure as a whole in order to be able to write monitors with correct behaviour. The following picture shows a simple outline of the SES components



As shown in the picture, the SES infrastructure actually consists of two relevant components:

- "SES (Secure Entry Server), a native reverse proxy, also referred to as "HSP" (for "Http Secure Proxy")
- "SLS (Secure Login Service), a Java-based authentication service

The HSP reverse proxy controls the HTTP or HTTPS connection with the client. If the HSP decides that a client needs to authenticate, it will prevent all requests from that client from reaching the application servers, and will instead redirect it to the SLS for authentication.

The SLS is involved only during the authentication step. It is responsible for displaying the login page and performing the authentication against the authentication backend, such as LDAP, RADIUS etc.

Therefore, a restart of an SLS instance will only lead to a short (\~ 1 minute) failure of authentication functionality, but it will not have any impact on existing application sessions.

1.2 SLS / Tomcat "Instances"

In most cases there will be just one SLS "instance" on a given server. But sometimes there might be more than one, for example one SLS "instance" for a Test-Environment and one for production use, or one for form-based login and a second one for NTLM authentication.

But what is an SLS "instance" anyway?

The SLS itself is just a Java web application that needs a servlet container to run it. By default, Apache's Open Source implementation "Tomcat" is the container used to run the SLS application. Tomcat itself is typically installed in a default location in the system, such as

```
/opt/apache/tomcat-6.0.18
```

This installation directory contains all Tomcat binary files (.jar files with the Java executable code) and base configurations.

A Tomcat "instance" is basically a copy of that directory tree, but without all the binary files, and with only the configuration files that differ from the defaults. Which HTTP port an instance uses is configured in that instance, for example. A typical Tomcat instance contains these directories:

```
/instance/
/bin/
/startup.sh
/shutdown.sh
/setenv.sh
/conf/server.xml
/temp
/work
/webapps/sls/...
```

Some of these directories often contain a lot more files, especially the "`bin`" directory, but the files mentioned above are the ones really relevant.

Note that the following two directories both contain only temporary data which can safely be deleted when an SLS is stopped and before it is started again:

```
/temp
/work
```

But it is of utmost importance that these two directories itself are present at startup time, or the Tomcat start will fail with sometimes unpredictable behaviour.

1.3 Tomcat Configuration

The following Tomcat configuration files are the most important ones for dealing with SLS operation problems:

`bin/setenv.sh`

This script, if it exists, is sources automatically by the other scripts. By defining the environment variables "`CATALINA_HOME`", it allows to define which Tomcat installation to use as the base for this instance. The variable "`CATALINA_BASE`" must contain the absolute path of the instance directory itself, or startup and shutdown will not work.

`conf/server.xml`

This file contains a local shutdown listener port, which must be unique within the local system, the HTTP listener port which must correspond to the port configured in the HSP / SRM login location configuration, and the context path under which the SLS web application will be accessible.

The contents of this file are important also to determine the correct URI path and port for performing a health check as described in "3 Health Check".

Chapter 2

Performing Changes

If changes are performed for an SLS instance, a few things should be considered and verified in order to be sure that the functionality is not broken:

2.1 Restarting an SLS

There is no standard script for SLS Tomcat instances. In case of a "standard" Tomcat instance, shutdown and startup are performed using the respective scripts in the "`bin`" subdirectory of the SLS instance:

```
%> bin/shutdown.sh
%> bin/startup.sh
```

If your environment uses different scripts (such as customized init scripts) please refer to your local runbooks for information on how to stop / start / restart SLS instances.

Note
A simple restart of the SLS instance will start the Tomcat / Java process, but not yet the actual SLS web application. To enforce a start of the SLS application a functionality check should be performed as described in

2.2 Backup / Rollback

Before any changes are performed, always make sure that a rollback to the last working state is possible. If all else fails after the change was applied, perform a roll-back and restart the SLS instance.

2.3 Checklist After The Change

- Check the file permissions (see "File Permissions")
- Leave no "dead" files/directories in "webapps" (see "No Dead Files in webapps")
- Restart the SLS if necessary (see change instructions)

- If possible, perform a local health check using "elinks" (see "3 Health Check"). If a login page appears (or the corresponding appropriate respond is received), the SLS instance at least started. If a page is returned with a "System Error" message, or a "404" or a similar Tomcat error, there was a serious problem and the SLS web application did not start correctly. Check the log files to find the cause for the problem (see for details). Fix them and restart the SLS instance again.
- If there are problems that cannot be resolved and a rollback must be performed, collect as much information as possible first for further analysis (log files and copies of the faulty configuration)

Chapter 3

Health Check

If possible, a health / functionality check should be performed after a change was applied to an SLS instance. To do this, the SLS should be contacted using a web browser or some other HTTP client:

Regular web-browser

If possible, connect with a regular web browser, just as the actual users will do it.

Commandline-browser

If you cannot connect to the SLS through a regular web browser from "outside" (through the reverse proxy), use an local shell-browser like "elinks" or "lynx" and connect to the Tomcat instance locally. This still allows to perform a simple login in most cases.

WGET

If no browser is available at all, use a tool like "wget" to at least trigger a start of the SLS web application and check the login page for any error messages.

Possible levels of a health-check are:

1. Send a request just to trigger the start of the SLS web application, and then check the log files for errors
2. Check the actual HTML page returned in the response for error messages
3. Perform a login to make sure that the SLS is not only running, but can actually perform its most important functionality. This depends on having valid login credentials, of course.

Once a request was submitted to the SLS instance, the SLS web application will be started by the Tomcat container. If there are any serious configuration problems, the SLS web application will either not be started at all, resulting in a "404 resource not available" response from Tomcat. Other types of SLS configuration errors may result in a "System Error" response page sent from the SLS. This indicates that the SLS is not ready for operation, even though the application was started. Check the "`exception.log`" and/or "`sls.log`" files for information on what might be the cause.

3.1 Regular Web-Browser

With a regular web-browser, the actual login URL depends completely on the login location configuration in the reverse proxy. Consult the HSP / SRM configuration to find the appropriate URL for the SLS.

3.2 Commandline-Browser

When using a commandline-browser, the local Tomcat HTTP listener port must be used and the context-path as defined in the Tomcat instance. Take a look at this file in the Tomcat / SLS instance:

```
%> cd <sls-instance-directory>
%> more conf/server.xml
```

An very minimalistic example result would be:

```
<Server port="4401" shutdown="SHUTDOWN">
<Service name="Catalina">
<Connector *port="4490"* maxHttpHeaderSize="48000" />
<Engine name="Catalina" defaultHost="localhost">
<Host name="localhost" appBase="webapps" >
<Context *path="/webapp/sls"* docBase="sls" allowLinking="true"/>
</Host>
</Engine>
</Service>
</Server>
```

NOTE: In "real-life" Tomcat setups, there might be a lot more configuration tags in this file, like log-"Valve"s etc. But the extract above contains only the most relevant parts.

The interesting bits are the "port" and the context "path" values. For the example above, the command to retrieve the login page with "elinks" would be:

```
%> elinks http://localhost:_4490/webapp/sls_/auth
```

The "action"-URI "/auth" must be appended to the context path. In case of some "older" SLS installations, the action may also be "login" instead of "auth":

```
%> elinks http://localhost:_4490/webapp/sls_/login
```

3.3 WGET

For a test with WGET, just issue a request to the local SLS Tomcat instance. Follow the descriptions for the commandline-browser in "Commandline-Browser" to find out the listener port and URI path, and then use the appropriate command:

```
%> wget http://localhost:4490/webapp/sls/auth
```

By default, this will create a local file named "auth" containing the HTML page received in the response. Check the page to see if it's a login form page, and if it contains any error messages.

Chapter 4

Monitoring The SLS

4.1 Introduction

In every modern IT environment it is common practice to check for the availability of important services in regular intervals. This is usually done by use of automated monitoring tools which perform some form of communication with the service in question.

4.2 Monitor Types

There are often two different classes of monitors (independent of the technology used for them):

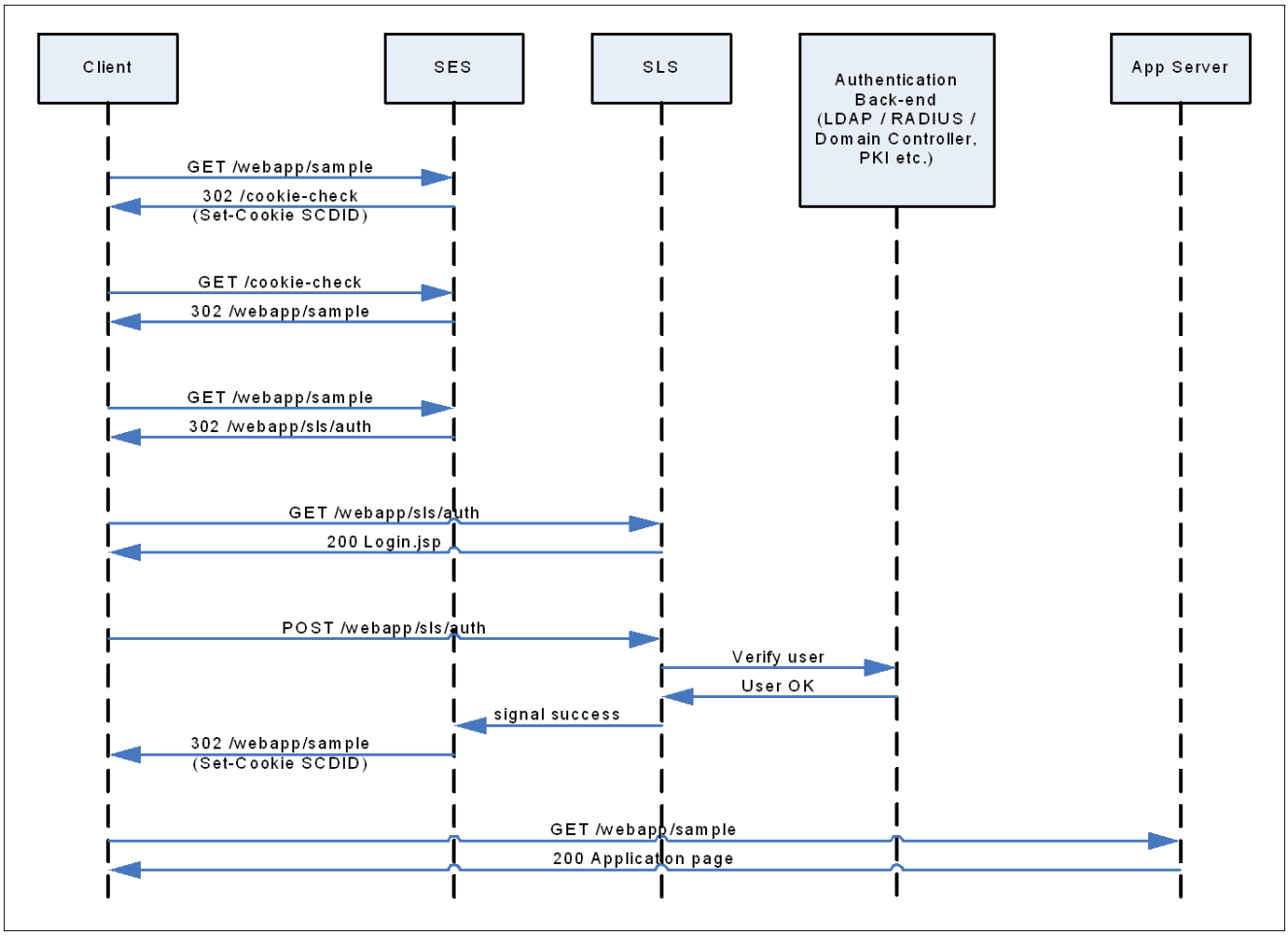
- Application monitors
- Infrastructure monitors

While the formers are meant to check for the availability of the actual application, the latter have to monitor components such as the HSP or the SLS (which are of course part of the infrastructure).

However, since applications protected by the HSP cannot be accessed directly and without an appropriate authentication step (as long as they are not defined as "public" in the HSP configuration), even application monitors need to interact with the Secure Login Service to some degree. At the very least they must be able to perform a login (usually using credentials of some technical test user), in order to get clearance to access the application server.

4.3 Login Process Example

A typical example of this process when trying to access a web application through a browser looks like this:



4.4 "SCDID" Cookie

When a client connects to the SES for the first time during a session, the SES always creates a cookie named "SCDID". This cookie is used only to map the client to back-end connections. It does NOT in any way indicate that the client is authenticated; it is just a handle for mapping connections from the same client. Therefore, any client must be able to support cookies in order to work successfully with the WES infrastructure (which means it must send the cookie back to the SES with all following requests).

Please note: This cookie is ALWAYS created by the SES (and NOT the SLS), not matter what the target destination of the client's first request was. So, no matter if the client sends a GET request to the application first, or a POST request to the SLS location, the response will always contain a newly created SCDID cookie from the SES.

Once the SLS signals a successful authentication, the SES re-creates the SCDID cookie.

If HTTPS is enabled, the cookie usually is named "SCDID_S". The name of the cookie can be changed in the SES configuration.

4.5 Login Service Response

A monitor should ignore most parts of the response of an SLS. For example, it is strictly recommended to avoid parsing any page contents, HTTP response code (302 or 200), HTML titles etc., and basing any success or failure rules on such contents.

The reason is that the behaviour of the SLS is always subject to change in future releases, also due to page redesigns within the customer's intra- or Internet site, or changing requirements of the environments etc.

Header "SLSStatus"

For this reason, the monitor should look for exact only one attribute of the SLS response, the custom HTTP header "SLSStatus". This response header is set by the SLS for one purpose only: To indicate the authentication status to any non-browser client, independent of all other aspects of the response. So for the monitor the important points are:

- Does the response contain a header "SLSStatus"? If yes, the response was actually coming from a Login Service. If not, the response was from another component, probably the HSP itself (like a HSP error page).
- If the "SLSStatus" header contains the value "200", the client was successfully authenticated. In this case, the client can now access the application. If the "SLSStatus" header contains any other value, the login failed.

4.6 Transparent Monitor Redirect Handling

If the response from the SLS hangs or shows some unusual delays, the problem can be either with the SLS, or with the monitor behaviour / configuration, in certain cases.

Example scenario:

1. "The application server has technical problems, and all responses from the application have a 1 minute delay.
2. "The monitor sends the POST login request to the SLS, expecting an immediate answer. Instead, the login response seems to hang for 1 minute.
3. "The monitor now shows a red flag, thinking that the SLS is hanging, which in this case probably isn't true.

Since the SLS does not access the application in any way whatsoever, this kind of problem usually points to the monitor's client code following the redirect response from the SLS automatically. So, in the monitor's code (Java, PERL script etc.), there may be just one single line of code for sending the POST request, but in some cases it depends on the arguments or configuration how the client will deal with a redirect response from the SLS. If the client just tries to transparently follow the redirect, it will of course experience the delay from the application server, and in the monitor's report, this will appear like a delay from the login POST.

4.7 Login Service Types

Please note that there are two fundamentally different types of login services:

- Form-based logins
- Transparent (SSO) logins, usually NTLM

For the first one, any monitoring client just needs to be able to send HTTP POST requests with parameters in order to interact with the login service.

The other kind of login service requires the monitor to perform proper NTLM (or Kerberos, certificate etc.) authentication, which is sometimes handled transparently by underlying technologies, such as NTLM-support in Java 5 for Java-based monitors.

4.8 Application end-to-end monitors

An end-to-end ("e2e") application monitor should reduce its SLS handling to the absolute minimum. First of all, it is not the obligation of such a monitor to perform in-depth functionality checks on the SLS. And in addition, there's always a possibility that the interface or behaviour of the login service might change in future releases; and in such a case, reducing the interaction with the SLS also reduces the risk of needing to adapt/change a monitor for a new SLS release.

A typical scenario for an e2e monitor could look like this:

1. Send a POST login request directly to the login location, with the appropriate parameters. Note that if a client sends a POST request directly (with no GET request first), the SES will not perform the cookie check (which means there will be no redirect to the "/cookie-check" location).

SCDID Cookie

1. In any case the response will contain a "Set-Cookie" header for the "SCDID" cookie. This header was not set by the SLS, but the SES (since this was the first request from the client in this session).
   The client must store this cookie and send it with any following request, just like a browser client would. +
2. Check if the response for this POST request contains a header named "SLSStatus". If not, the response is not even from the SLS, or there is a severe technical problem. In either case, the login attempt has failed.
3. If there is a HTTP Header "SLSStatus", check its\' numerical value:
   If it is "200", the login was successful. From this point on, the monitor can access the application.
   If it is "403", access was denied (authentication failed).
   If it is "500", there was a technical problem within the SLS.
4. In addition, the monitor can send (after a successful login) a first request to the application and check if the response is the application page. If it is instead a HTTP 302 (Redirect) back to the login page, there is still a problem with the authentication or entry server configuration.

Chapter 5

FORM-based Authentication

For performing an actual authentication against an SLS with FORM-based login, the monitor must send a POST request to the SLS. It is recommended to make attributes like the request URL, parameter names etc. configurable in order to be prepared for any possible future change in the login service.

5.1 Example

In this example of the ficticious company "ACME Inc.", their login service had been configured to be available under the URL context

```
/webapp/sls
```

being the base context of the login service, so that the actual login action would usually be

```
/webapp/sls/auth
```

So, the whole URL to POST the request to would be:

`http://www.acme.com/webapp/sls/auth`

The required request parameters, based on the names of the form fields in the template login JSP included with the SLS delivery:

Name: userid

Value: The user’s login ID

Name: password

Value: The user’s login password

Name: formName

Value: loginForm

Chapter 6

NTLM Authentication

The Windows SSO login service uses NTLM to transparently authenticate the user through his web browser, using the Windows login credentials. Any monitor trying to access an application that uses NTLM authentication must support this authentication scheme itself.

- NTLM Information:
  `http://davenport.sourceforge.net/ntlm.html`
- PERL NTLM support:
  `http://drupal.org/node/44718`

6.1 Java Clients and NTLM

For any Java-based monitor, there are a few relevant things to consider in relation to NTLM authentication:

- NTLM authentication is officially available on all platforms in Sun's "java.net" HTTP classes (since Java 6).

There are other client-side HTTP implementations which may provide their own level or type of NTLM support:

- Jakarta's Commons HTTP Client (open source)
  `http://commons.apache.org/httpclient`
- Oakland Software HTTP Client (commercial)
  `http://oaklandsoftware.com/product_http/overview.html`

If the technology/API used by the monitor is based on Sun's HTTP implementation in the JDK, the NTLM authentication is basically available for free. All "HttpUnit"-based clients for example (see `http://httpunit.sourceforge.net`) can just directly send their requests to the application, since the NTLM authentication is handled transparently by the underlying HTTP implementation. This means that there is no need to send any request to the login service; that all happens "magically" within the HTTP implementation.

6.2 Login Credentials

Please note that, on Windows, the NTLM implementation in Sun's JDK code will always perform a login with the Windows credentials of the current session (that is, the session which the code is running in) first. Only if that login fails will it use the call-back interface

```
java.net.Authenticator.getPasswordAuthentication()
```

for retrieving the user credentials. By providing a custom Authenticator subclass implementation, specific credentials can be provided for the next login attempt.

On other platforms like UNIX, the Authenticator interface is used also for the first login attempt.

Chapter 7

Logging And Debugging

7.1 SLS / Tomcat Log Files

By default, the SLS creates a number of log files in the subdirectory "logs" of the SLS instance. Some log files are created by the SLS application, and others are created by the Tomcat container.

SLS log files:

`audit.log`

Contains one log entry for each authentication attempt (success / failure) or operations such as password changes etc. Note that in case of a 2- or 3-step-login procedure, this log file still gets only one log record for the whole login process, not one per request.

`sls.log`

The SLS debug log file. If debug or trace logging is enabled, this file will contain a lot of information useful to find the source of any problem.

exception.log - Contains the error logs in case of serious internal errors, and also the corresponding Java stacktrace, if there was one. This is information that can be very useful for the USP 3rd-level support for deeper problem analysis.

`performance.log`

Contains one log record per request and allows to locate - to a certain degree - the point where a performance problem lies (see).

`catalina.out`

Contains the "stdout" and "stderr" output of the SLS Java process. In some cases, error information is printed only to this log file (especially if a problem occurred in a 3rd-party component that is not directly part of the USP Java code). So in case of problems, the contents of this file should be collected for 3rd-level support as well.

Any additional files are depending on the Tomcat configuration as defined in the "conf/server.xml" files. Please consult the Apache Tomcat documentation for more information:

http://tomcat.apache.org/tomcat-6.0-doc/logging.html[http://tomcat.apache.org/tomcat-6.0-doc/logging.html]

7.2 Log aggregation / correlation

All requests that pass the SES Secure Entry Server will recieve a so-called "client-correlator" and "request-correlator" header. As the names imply, the client correlator value allows to correlate log entries from different log files to the same client session, and the request correlator allows to do the same for one single request.

For this reason, all log files created by the SLS application contain a `[CC:<client-correlator]`-string and an `[RC:request-correlator]` string.

If there is a complex problem that may involve the reverse proxy as well, collect all logs from the SES log files and the SLS log files for the client session where the problem occurred by grepping the log files for the entries with the corresponding correlators. In case of the SLS alone, this might be useful to extract only the interesting data from the "`sls.log`" file.

A typical example would be:

- A customer says that certain logins are failing for unknown reasons.
- If debug logging was not yet enabled, enable it and ask the customer to perform the login again in order to collect come data (see "Debug / Trace Logging" for information on how to enable debug logging).
- Ask for the user ID in question and at what time the failed login was performed.
- Search the "`audit.log`" file for some "authentication failed" message for the user and time given by the customer. This is easier in the "`audit`" log file because it usually only contains one log entry for a complete authentication procedure.
- Then copy the "CC:"-value from the "`audit`"-log entry and use it with a "`grep`"-command to extract the corresponding debug log entries from the "`sls.log`" file.

7.3 Debug / Trace Logging

If there is a problem that can be reproduced, it may help to collect debug information that can be forwarded to 3rd level support if necessary, or just to find the cause for the problem and fix it.

The so-called "debug" log file is usually named "`sls.log`". In production environments, it contains only messages of level INFO or above, so the logging configuration must be changed to activate debug logging.

The configuration file that must be changed is:

```
<sls-instance>/webapps/sls/WEB-INF/log4j.xml
```

In it, there are both log "`appender`"-tags and "`logger`"-tags. To activate maximum debug (or trace) log level, change the value of the "level"-tag of the "`com.usp`"-logger to "DEBUG" or "TRACE", for example:

```
<logger name="com.usp">
**              <level value="DEBUG" />**
**      <appender-ref ref="SLS_LOG" />
<appender-ref ref="EXCEPTION_LOG" />
</logger>
```

- `DEBUG` produces a lot more information about the internal processes, but still on a maintainable level
- `TRACE` logs basically ever internal method call and exit; this level is usually only useful if the log information must be forwarded to USPs 3rd-level support for deeper analysis.

NOTE: Do NOT change the "Threshold"-values defined in the various "<appender>"-tags. They should always remain unchanged! Only change the "<level>"-value of the "<logger>"-tags!

Restart not mandatory

Log4j automatically re-reads the "`log4j.xml`" configuration file after a while, so a restart of the SLS instance is not mandatory. However, note that it can take up to a minute until the configuration becomes active.

what is "<category>"?

In previous Log4j releases (the logging API used by the SLS), the "<+logger>+" tag was named "<+category+>", and the "<+level>+" tag was named "<+priority>+". Since Log4j is still backwards compatible, both can still be used; but it is recommended to replace "<+category>+" with "<+logger>+" and "<+priority>+" with "<+level>+" in older configuration files.

7.4 Error Messages

When there is an SLS error, the "`exception.log`" and sometimes also the "`sls.log`" file get one or more log records on level ERROR, like this example:

```
2009-03-13 17:36:44,126 [CC:...] [RC:...] - [AUDIT] [*USER_AUTH_FAILED_TECH*]  ↩
    Authentication failed due to a technical problem. User: \'demo\'. Reason: \' ↩
    Authentication for user \'\'msc\'\' failed, reason: Receive timed out\'
```

This is an example of a RADIUS authentication that failed because the RADIUS server became unavailable, and the connection attempt from the SLS to the RADIUS server timed out.

The highlighted string `USER_AUTH_FAILED_TECH` is the SLS error message ID that identifies this kind of error. All SLS error messages are documented in the separate "`log-messages.pdf`" file. Please read that documentation for more information about any given error, its possible causes and what to do.

7.5 Performance Logging

There is a performance log file which receives one log entry for each request the SLS processes. It contains measurements of the internal steps that were performed by the SLS, and of the call-outs.

A typical entry in an LDAP login service might look like this:

```
2009-03-13 ... - [Request: 505 [do.auth: 449 [ldap: 446] ] [do.success: 56] ]
```

The most interesting part is usually the step referring to the back-end call-out, which is the "[ldap:]"-part in this case. A short run-down of the example and its parts:

`Request: 505`

means that the entire request took 505 milliseconds to be processed by the SLS.

`do.auth: 449`

means that the "`do.auth`"-step (the actual authentication) took 449 milliseconds to complete. This includes both SLS-internal functionality as well as the call-out to the authentication back-end system.

`ldap: 446`

means that the actual LDAP processing took 446 milliseconds. Note that this step is nested inside the "`do.auth`"-step, because the call-out to the LDAP service is a part of the authentication step. As a result, the log data means that the entire authentication step took 449 milliseconds, but only 3 milliseconds were spent in the SLS internal logic, and 446 milliseconds were spent performing the LDAP call.

This is usually the most important point to check when there are any performance problems. If the call-out takes a long time, the problem is probably related to the back-end system and not the SLS itself.

`do.success: 56`

This is the final step where the SLS creates any login tickets, custom headers, cookies etc. and performs the redirect to the application. Depending on the configuration and on what the SLS has to do in this step, this can also be a more expensive step. For example, if encrypted and signed tickets must be created after a login, the "do.success" step will take considerably longer because encryption is a very CPU-intensive type of operation.

Chapter 8

Debugging / Troubleshooting

8.1 File Permissions

Typical symptom: "System Error" after SLS restart

A typical source for problems is that changes are performed manually using an account such as "root", but the SLS service running with a more restricted account. If new files or directories are created during the changes, they will belong to "root", and depending on the local system setup, the SLS service user may not be able to read those files.

Check the local SLS solaris package to see what the correct permissions should be. As a general rule of thumb, make sure that

- the SLS service user has read access to all directories of the Tomcat instance
- the SLS service user has write access to the the log directory and all log files.

In most cases missing read-permissions are the cause for problems.

8.2 No "Dead Files" in "webapps"

Typical symptom: Mysterious errors, new functionality missing

Make sure that the "webapps" directory of the SLS instance contains only the directories it is supposed to contain; usually, this is just one subdirectory with the SLS web application. In some cases there might also be a second web application with "static" files such as images, CSS files etc.

A typical cause for "dead" web SLS copies in the "webapps" directory are manual backups made before a change was applied. Never leave backup copies of the SLS web application in the "webapps" directory!

8.3 Tomcat Temporary File Issues

Typical symptom: Mysterious errors, new functionality missing

There are rare cases when a Tomcat instance for some reasons does not update the temporary files in its "`work`" subdirectory correctly after changes have been applied. If there are strange problems such as new functionality not being available even though a new software release had been applied, delete the *contents* of the "`work`" directory and restart the SLS instance:

```
%> rm -rf work/Catalina
```

Note
Do NOT delete the "`work`" directory itself, but all of its contents!

Restart the SLS instance afterwards. It is normal that the first requests will be slower in this case, because Tomcat needs to re-compile all Java Server Pages (JSPs) again.