C++ Developer's Guide

Subversion

The SP is split across three primary projects, **cpp-xmltooling**, **cpp-opensaml**, and **cpp-sp**. At any given time, development is occurring on one or more branches, but not currently the trunk. Each project is a self-contained automake/autoconf/libtool project (for non-Windows) and a Visual Studio C++ solution containing various subprojects (for Windows). Each project builds on the next, so first dependencies are built and installed, then xmltooling, opensaml, and finally the SP.

Releases within a major version and the latest minor version are in a branch called ***M.x*** where ***M*** is the major release. For example the 2.x SP code is in the **2.x** branch.

Releases within a previous minor version (patch releases) are in a branch called ***M.N.y*** where ***M*** is the major release and ***N*** is the minor release. For example, the 1.4.x xmltooling code is in the **1.4.x** branch.

Build Environment and Compiler Notes

Non-Windows

The subversion checkout is in an "unbootstrapped" state and building requires the complete set of autotools (automake, autoconf, libtool). With modern autotools, running `autoreconf -if` is sufficient to bootstrap the process. There should be few if any warnings, so noisy output indicates an environmental problem. To generate the "cleanest" (meaning the most portable) scripts and makefiles, FreeBSD is ironically the best place to do the bootstrap and generation of a distribution file. Using Solaris is a lot of work to get setup, and not terribly portable, and using Linux tends to introduce Linux-only shell script features. The BSD results seem to have the best chance of working on all the officially supported platforms.

For all but Solaris, the platform default GNU compiler collection should work fine. Mac OS X is a bit more complex in that the built-in compiler is based on gcc + llvm, while the macports default compiler is clang. The latter is usually more strict, so just building from the shell isn't sufficient to verify the build.

Solaris is a nightmare. Rather than reproducing all the details, see NativeSPSolarisSourceBuild.

Windows

See also NativeSPWindowsSourceBuild.

The Windows projects are self-contained and are usable immediately after check-in. Each project is a Visual Studio solution file containing various subprojects. The current builds are being done with the VS2010 build chain, but can be performed with either VS2010 or VS2012 by changing the build tools in the project files. Each project includes twp property sheets which establish the link and include paths, based on the current versions of the dependencies. This sheets are checked into the cpp-xmltooling project and included as svn:externals in the other two solutions.

The expected layout of a Windows build is to place all the dependencies in a common directory, and then checkout or unpack the Shibboleth sources together in a common directory. To build the various Apache modules, installations of Apache should be unpacked to the root of the drive. The newer versions can be obtained from http://www.apachelounge.com/. Each project includes a further property file called `BuildPaths.props`, which you have to place in the directory above the solution directories. This has to define a single property, `BuildRoot:`

Example BuildPath.props

```
<Project DefaultTargets="Build" ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
        <PropertyGroup Label="PathMacros">
                <BuildRoot>C:\perforce\ShibbolethBuild</BuildRoot>
        <Apache13Root>C:\Apache</Apache13Root>
        <Apache20Root>C:\Apache2.0.63\Apache2</Apache20Root>
        <Apache22Root>C:\Apache2</Apache22Root>
        <Apache24Root>C:\Apache24</Apache24Root>
        <Apache22Root64>C:\Apache2-64</Apache22Root64>
        <Apache24Root64>C:\Apache24-64</Apache24Root64>
        </PropertyGroup>
</Project>
```

You should not have to make any changes to the files contained in "%APPDATA%\Local\Microsoft\MSBuild\v4.0\" and should revert any changes made there for previous builds.

Release Engineering

The release process for the SP involves a number of steps that can be performed in a relatively loose order, but experience has led to the following outline to minimize the chance of overlooking a portability problem and having to repeat steps. In general, a failure in any of the steps means starting over again, or there's a good chance of introducing a regression on one of the platforms.

The basic summary follows, and then each step is elaborated.

1. Generate source distributions for use by the later steps.
2. Test Solaris, Mac, and macport builds.
3. Do the Windows build and installers.
4. Generate prerelease "final" RPM packages.
5. Upload final source distributions and Windows installers to public site.
6. Generate official RPM packages.
7. Update the macports.

Source Prep

The first step is to prepare candidate source releases. FreeBSD is recommended for this purpose as discussed earlier, because it seems to produce the best autoconf results for the most platforms. Using a FreeBSD VM:

1. Ensure all the dependencies are available.
2. Pull the candidate sources from subversion.
3. In sequence, bootstrap each project with `autoreconf`, then run the configure, make, make install sequence.
4. After fixing any build issues, prepare each of the distributions with make dist.

Solaris Smoke Test

It's a good idea to test periodically, but no matter how small the changes are, you need to run a final check of the source that's going to get released.

The reality of this platform is that you'd have to test all of the combinations (Sparc and Intel, 32- and 64-bit, g++ and CC) to know for certain that there won't be a problem, but that's not really practical at this point. Most software on Solaris is built as 32-bit code regardless of the kernel, so other than verifying that specific reported problems are fixed, the priority is to test with the Sun compiler in a 32-bit build on at least one and preferably both archs.

Functional testing here is a nice bonus, but the priority is that it builds. Running at least the unit tests for xmltooling and opensaml is a good idea of course.

Mac OS X Smoke Test

After Solaris, the next most likely source of regressions will be OS X, so a build there is the next step. After a build from the console, it's a good idea to actually do a macport test because they use the more strict clang compiler now. Local port testing is fairly obnoxious but a straightforward way is as follows:

1. Place the source distributions on an accessible web site.
2. Move to **/opt/local/var/macports/sources/rsync.macports.org/release/ports**
3. Modify the installed Portfiles beneath these folders. Make sure to update versions and the checksums.
4. Run `portindex` against the modified tree.
5. Test port upgrade or installs of the candidate portfiles.

Windows Build

The Windows build tends to be in better shape if its the primary development platform, but for other developers this may not be the case. It may be advantageous to move the Windows build earlier to catch regressions there, because any fixes end up risking regressions to the rest of the builds, so the goal once this step is reached is for the chance of regressions to be low.

Preferably long before this point, but as a final check, make sure that all of the Windows resource files in the projects have been updated with final version numbers for the release, including the Windows installer files.

The "final" build is done on a VMWare virtual machine (currently running Server 2008 R2) that has the tools installed. TortoiseSVN is installed there to allow the subversion code to be checked out directly. The builds are done out of the Administrator account, with all the dependencies and source inside **C:\Users\Administrator\Shibboleth**

Once all the dependencies are in place, building the three top-level solutions is just a matter of running the builds in each solution in the IDE for all four configurations (Debug and Release for both x86 and x64). With VS2010, at least, "batch" builds of all the configurations at once simply does **not** work. You literally get the wrong builds done in the staging directories and the link fails. I haven't tested VS2012 there yet, but the safest thing is just the slog of building each solution manually four times.

Once the builds are done, the installers have to built. These files currently live in **cpp-sp\msi\WiX**. The **Versions.wxi** file contains the primary version information for the release. The installer filenames, which can always be manually changed post-build too, are set in **Compile.bat**

The installer build process is currently run with a pair of batch files that generate the merge modules for all the libraries and schemas, and then the actual SP installers. First the merge modules are built using **cpp-sp\msi\WiX\MergeModules\Compile.bat** and then the installers are built using **cpp-sp\msi\WiX\Compile.bat**

Linux / RPM Builds

The RPM packages are maintained in the OpenSUSE build service. Prerelease packages can be built and tested in the home:Scott_Cantor project. The official packages are done in the security:shibboleth project.

It appears to be the case (2013-12-02) that the build service's web site won't let you log in unless you have once used the command-line `osc` command line tool. No, that makes no sense.

Once the packages are published in the production project, they're basically official and public, so this is the **last** step and should only be done once the packages are built and tested in the prerelease project and the sources have been frozen and uploaded as official releases.

Each dependency and the three SP packages are loaded into packages in the build service project. Each separate version of a package is loaded into its own new package named with the version number. For example, xml-security-c 1.7.0 is built in a package called xml-security-c-1.7.0 (duh). This allows multiple versions to co-exist in the repositories so that newer releases won't blow away the older packages (so history is maintained and older versions can be manually installed).

Each package contains two files, the tarball containing the source distribution and the RPM specfile for the package. If there are any source patches required by the specfile (this is rare), they're also added. While the web interface can be used to create packages and such, once they're created the `osc` command line tool can be used to add and commit/update source files in the various packages. It can be obtained via macports or on a SUSE Linux machine.

For the builds to be successful, special project configuration settings are required. These are set from the command line or in the "Advanced|Project Config" option tab:

osc meta prjconf -e

```
%if 0%{?rhel_version} > 0
Release: <CI_CNT>.<B_CNT>%%{?dist}
%endif

%if 0%{?rhel_version} >= 600 || 0%{?centos_version} >= 600
Substitute: !libxerces-c-devel
%endif
```

The purpose of these settings is to append a distribution-specific tag (e.g. "el5") to the package names in the Red Hat platforms, and to turn off a package substitution rule on Red Hat and CentOS 6 that prevents us from using our updated xerces-c-3.1.1 package instead of the one that comes with Red Hat. This was a mistake; the built-in package should have been used, but because I didn't do that originally, upgrades won't work properly unless I keep using the newer version.

Other special build notes:

- Red Hat and CentOS platforms need to have the "Debuginfo Flag" setting turned off under "Repositories". If not, the builds will fail. I don't know why.
- The "xerces-c" package(s) are disabled on SLE and OpenSUSE V11 and later. (As noted above, this should have been done for Red Hat 6, but wasn't, and it's too late to change that without breaking upgrades.

Apart from all that, the builds should be incident-free by the time a release is being done, but there are usually glitches and issues to work out prior to the release being close to done. The specfiles do not come from the actual source distribution, so what usually happens is that once the sources are frozen, a change or two might be made to the specfiles within the build service to get the packages done. Those get checked back in eventually, but may not make it into the actual release being done at the time unless the sources need to be regenerated for some other reason anyway. (As noted earlier, usually this can't be done while the production packages are being generated.)

Note that any time a dependency gets rebuilt, anything above it in the chain will be rebuilt, so usually its best to make sure all packages are disabled and then re-enable them one by one to get a successful build done before moving on to the next package.

Release Publication

The initial set of release artifacts (and the appropriate release location) consists of:

- shibwww@shibboleth.net/home/shibwww/html/downloads/c++-opensaml/*ver*/xmltooling-*ver*.tar.gz
- shibwww@shibboleth.net/home/shibwww/html/downloads/c++-opensaml/*ver*/opensaml-*ver*.tar.gz
- shibwww@shibboleth.net/home/shibwww/html/downloads/service-provider/*ver*/shibboleth-sp-*ver*.tar.gz
- shibwww@shibboleth.net/home/shibwww/html/downloads/service-provider/*ver*/win32/shibboleth-sp-*ver*-win32.msi
- shibwww@shibboleth.net/home/shibwww/html/downloads/service-provider/*ver*/win64/shibboleth-sp-*ver*-win64.msi

ver is replaced by the version number. A symlink called *latest* is also reset in the two product directories to point to the updated version directory.

Each artifact is accompanied by a PGP signature (with ".asc" added to the filename) and an md5 checksum (with ".md5" added).

Once these are published, the macport Porfiles can be updated in the official macport tree safely, and the final RPM packages built and published (as described in the previous section).

If any showstopper bugs appear at this stage, they have to be addressed either with patches applied during the packaging process, or by issuing an updated patch release and essentially deprecating the release that's in progress. In other words, **don't** upload the sources until everything is 99.9% certain to be working. For sure, don't do it until RPM generation in a prerelease project is confirmed.

Once the RPM build process is completed, the SRPMs are copied into the SRPM subdirectory (a sibling of the win32 and win64 directories). The SRPMs can be copied from any of the build service repositories (all the SRPMs in the repository's src directory should be identical regardless of the platform).

Finally, the three projects are tagged in git to complete the process.