Vtable Verification User's Guide

[What is vtable verification?](#)
[How can you get a compiler with the vtable verification feature?](#)
[Flags](#)
[Trouble-shooting: Getting a Verification Failure](#)

What is vtable verification?

Vtable verification is a new security feature implemented in the compiler and controlled by a compiler flag ("-fvtable-verify=std" -- See Flags section for more details). The idea behind vtable verification is to validate the vtable pointer inside a C++ object immediately prior to the vtable pointer being used to make a virtual call. The compiler will detect if the vtable pointer value in the object is not valid for the type of the object, and if that is so, then it will abort execution (on the assumption that the vtable pointer has been corrupted, probably due to a malicious attack). If the vtable pointer value is valid, the execution continues as normal, and users should see no differences at all.

How can you get a compiler with the vtable verification feature?

Currently, to use vtable verification you will need to configure and build the compiler with vtable verification enabled. Assuming you have the compiler source tree (see the documentation under gcc.gnu.org if you do not), you can configure and build the compiler with:

```
$ ../gcc-top/configure  <your-normal-config-flags-here>
--enable-libstdcxx-threads --enable-vtable-verify=yes
$ make all
```

NOTE: If you might want to prefix the make with CFLAGS="-g -O0" and CXXFLAGS="-g -O0", as this will help with trouble-shooting, if there are any problems.

Finally, you will probably want to install the built compiler and its runtime libraries somewhere in order to be able to easily invoke it. For example:

```
$ make DESTDIR=/home/<user>/my-gcc-root install
```

Now you should be able to run the vtable verification compiler. Note that you must always explicitly tell it where the runtime libraries are that it should use (via the -Wl,-R flag, as shown below), or it will use the standard ones installed on your machine, which will not work with vtable verification. You also need to pass the "-rdynamic" and the "-Wl,-z,relro" flags, as well as "-fvtable-verify=std", if you want to use vtable verification. For more complete information on the flags you may want to use with the vtable verification feature, see the flags section below.

```
$ /home/<user>/my-gcc-root/usr/local/bin/g++ -fvtable-verify=std
-rdynamic -Wl,-z,relro -m64
-Wl,-R,/home/<user>/my-gcc-root/usr/local/lib64
<any-other-compiler-flags-you-want> your-program.cc

$ /home/<user>/my-gcc-root/usr/local/bin/g++ -fvtable-verify=std
-rdynamic -Wl,-z,relro -m32
-Wl,-R,/home/<user>/my-gcc-root/usr/local/lib32
<any-other-compiler-flags-you-want> your-program.cc
```

Flags

There are several flags and/or you may want to use with the vtable verification feature:

-fvtable-verify=std
-fvtable-verify=preinit
-fvtable-verify=none

The first two of these flags turn on the feature. The third flag disables the compile-time portion of this feature if, say, some other script has automatically added one of the first two flags to your compile line. If more than one of these flags occurs on the compile line, the compiler will use the last one it encounters as the "real" flag for this feature.

The difference between 'std' and 'preinit' has to do with when the runtime functions that build the verification data structures get run. Normally (using the 'std' version), these functions get run after the shared libraries for the program have been loaded and initialized, and before 'main'. If you need these functions to run BEFORE the shared libraries get loaded and initialized, then you should use the 'preinit' version.

IMPORTANT NOTE: For the vtable verification feature to work properly, the gcc/g++ driver must be used not only for the compilation phase, but also for the *linking* phase. When the compiler driver sees '-fvtable-verify=std' or '-fvtable-verify=preinit', it adds certain necessary things to the link line. Without these additions to the link line your program will get seg faults when you try to run with vtable verification.

-fvtv-counts

This is a feature debugging flag. If you use this flag, then during compilation the compiler will keep track of the total number of virtual calls it encountered and the number of verifications it inserted. It also counts the number of calls to __VLTRegisterSet and __VLTRegisterPair that it inserts, as well as the number of unused vtable map variables it created. This information, for each compilation unit, is written to /tmp/vtv_count_data.log. It also counts the size of the vtable pointer sets for each class, and writes this information to /tmp/vtv_class_set_sizes.log

Note: This feature APPENDS data to the log files. If you want a fresh log files, be sure to delete any existing ones.

-fvtv-debug

This is another feature debugging flag. If you use this flag during compilation, the compiler will keep track of which vtable pointers it found for each class, and record that information in /tmp/vtv_set_ptr_data.log.

This flag will also cause the compiler to generate calls to the debug versions of __VLTVerifyVtablePointer, __VLTRegisterSet and __VLTRegisterPair, rather than the normal ones (the debug versions output more information).

Note: This feature APPENDS data to the log file. If you want a fresh log file, be sure to delete any existing one.

-fdump-tree-vtable-verify

This is yet another debugging flag. It causes the compiler to dump the GIMPLE intermediate representation of the program, just after executing the vtable verification pass, to a file. This allows you to examine the verification calls that were inserted, the parameters passed to the verification calls, and the constructor initialization function that builds up the data set, for each compilation unit.

When **building** the compiler (not running it), there are several macros you can use to affect the vtable verification behavior:

-DVTV_NO_ABORT

If you use this macro, then when a verification failure occurs, the runtime library will report the error, (in /tmp/vtv_logs/vtv_verification_failures.log) but will NOT abort the program's execution.

-DVTV_STATS

If you use this macro, then when the program built with vtable verification is running, the runtime library will collect stats about how many calls are made to mprotect, __VLTRegisterSet, __VLTRegisterPair and __VLTVerifyVtablePointer, and how many cycles are spent in those functions. If you use the debugger on the running program, you can call __VLTDumpStats(), which will write out these numbers to /tmp/vtv_logs/vtv-runtime-stats.log.

-DVTV_DEBUG

If you use this macro, the runtime library will output a fair amount of debugging information while it is running. This works best when the program was built with -fvtv-debug as well. Output files include /tmp/vtv_logs/vtv_memory_protection_data.log, /tmp/vtv_logs/vtv_init.log, /tmp/vtv_logs/vtv_verify_vtable.log .

-DHASHTABLE_STATS

This macro will cause various statistics about the vtable map hash tables to be written to /tmp/vtv_logs/vtv_set_stats.log.

Trouble-shooting: Getting a Verification Failure

It is possible that you will build your program, run it, and get a verification failure (i.e. your program aborts). Now what? As explained at the start of this document, you will get a verification failure (and your program will abort), if the verification function looks for a vtable pointer in the set of valid pointers for an object's statically declared type, and fails to find the pointer in that set. A vtable pointer might fail validation for one of three reasons: 1), The value really is bad (due to a hacker attack); or 2). The C++ code contains an incorrect cast between two types (so the program was not attacked, but the vtable pointer really is not in the class hierarchy for the object); or 3). The data set used to construct the set of valid vtable pointers is incomplete. If you are running/testing a program you just built, then case 1 is not very likely. That leaves cases 2 and 3. Case 3 can happen if you are linking with and C++ code that was not built with verification, especially libraries and/or third party code (for a more in depth explanation, see the document Vtable Verification - Info for Third Party Developers .

How can you tell what is really going wrong?

- Rebuild your compiler in debug mode (if you didn't build it that way originally). You do NOT need to re-compile your program at this point. The reason for rebuilding the compiler is to get a debuggable runtime library .
- Start your program under gdb e.g.

```
$ gdb a.out
```

- Set a breakpoint at __vtv_verify_fail. The debugger will want to create a pending breakpoint, because __vtv_verify_fail is actually defined in a shared library which has not been loaded yet:

```
(gdb) b __vtv_verify_fail
Function "__vtv_verify_fail" not defined.
Make breakpoint pending on future shared library load? (y or
[n]) y

Breakpoint 1 (__vtv_verify_fail) pending.
(gdb)
```

- Run until you get to that breakpoint:

```
(gdb) run
Starting program: /home/cmtice/vtable_tests_git/a.out
Breakpoint 1, __vtv_verify_fail (
   data_set_ptr=0x603008 <_VTV<derived>::__vtable_map>,
   vtbl_ptr=0x601d70 <vtable for base+16>)
   at ../../../../gcc/libstdc++-v3/libsupc++/vtv_rts.cc:1512
1512             is_set_handle_handle (*data_set_ptr) ?
```

- Examine the input parameters to see what vtable and set were being used. This will ONLY work if the runtime library was built to be debuggable:

```
(gdb) x/x vtbl_ptr
0x601d70 <_ZTV4base+16>:   0x00401260
(gdb) x/x data_set_ptr
0x603008 <_ZN4_VTVI7derivedE12__vtable_mapE>:   0xf7fe6040
```

- Do a backtrace, to see where the virtual call was that caused the problem:

```
(gdb) bt
#0  __vtv_verify_fail (data_set_ptr=0x603008
<_VTV<derived>::__vtable_map>,
   vtbl_ptr=0x601d70 <vtable for base+16>)
   at ../../../../gcc/libstdc++-v3/libsupc++/vtv_rts.cc:1512
#1  0x00007ffff7a96611 in __VLTVerifyVtablePointer (
   set_handle_ptr=0x603008 <_VTV<derived>::__vtable_map>,
   vtable_ptr=0x601d70 <vtable for base+16>)
   at ../../../../gcc/libstdc++-v3/libsupc++/vtv_rts.cc:1353
#2  0x0000000000400f93 in main () at temp_deriv3.cc:75
```

In this case, the problem virtual call was in the file temp_deriv3.cc at line 75.

- Exit from the debugger and use C++filt to figure out what vtable and class were being verified (based on the results of examining your parameters, two steps back):

  ```
  $ c++filt _ZTV4base
  vtable for base
  $ c++filt _ZN4_VTVI7derivedE12__vtable_mapE
  _VTV<derived>::__vtable_map
  ```

  So in this case our failure was caused by searching for the vtable for class ‘base’ in the set of valid vtable pointers for class ‘derived’. I.e. we are expecting ‘base’ to be a sub-class of ‘derived’. At this point it’s time to check the source code and see if ‘base’ is defined to inherit from ‘derived’ or not. If not, then you have a bug in your code and you need to figure out why an object with the dynamic type of ‘base’ was found in an object that was declared to be of type ‘derived’ at line 75 of temp_deriv3.cc.

  On the other hand, if ‘base’ does inherit from class ‘derived’, then you need to find the file that declares this to be the case, and see if that file was compiled with vtable verification turned on. If not, then compile it with vtable verification.

  If ‘base’ does inherit from ‘derived’ AND the file that declares this was compiled with vtable verification, you may have uncovered a compiler bug and it is time to create a bug report. This is the least likely scenario, however.

TroubleShooting: Getting a Seg Fault

As mentioned earlier, when using vtable verification, you must use the gcc/g++ driver not only for the compilation phase, but for the linking phase as well. If you generate your .o files at one time, and later link them together to create the binary you must make sure that you use the gcc/g++ driver; and you must make sure that you pass the same -fvtable-verify=... flag to the driver at link time that you did at compile time.

If you build your program with vtable verification and then get a seg fault when you try to run it, the chances are high that you did not link it properly. The seg fault usually means you are attempting to update the vtable data sets, and the protections on the data are still read-only (which will occur if you did not link properly).