# Vtable Verification: Verifying the Whole World

"Vtable Verification" is a new security feature being added to the compiler. The idea behind this feature is to check that the value of the vtable pointer in an object is valid (i.e. has not been corrupted or attacked), before the pointer is used to execute a virtual function call.

In order for this security feature to work properly, is it important that when the verification gets turned on, *all* of the C++ code for the program (including third party libraries and plugins) has been compiled with the vtable verification feature turned on, if this is possible. (See below for a more detailed explanation as to why this is so.)

## More details about how this works and why everything needs to be recompiled with this new compiler:

A virtual method call (in C++) is a function call that is made through a function pointer. The function pointer is stored in a table of function pointers for the class (the "vtable" for the class). Each object for a class contains a pointer to the vtable it should use for virtual function calls. The validation is done by comparing the actual vtable pointer value in the object against a set of known valid vtable pointer values, for objects belonging to the current object's class (there may be multiple valid vtable pointer values because of class inheritance). The compiler builds up these sets of valid vtable pointers in each object file during compilation, then aggregates all the data into complete sets at the beginning of program execution.

If the validation fails, i.e. the vtable pointer is not found in the set of valid pointers, an error message is emitted and execution (of the program) is halted. 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. The last case can occur if some of the files that define the important classes are compiled with vtable verification, and other files that contain part of the class hierarchy are compiled without vtable verification. A similar effect also occurs with libraries and/or plugins which pass objects (which make virtual calls) across their boundaries, if the library or plugin was compiled with vtable verification and the code on the other side of the boundary was not, or vice versa. This is why it is important that *all* C++ code in the program be compiled with vtable verification if at all possible.

## Example of Potential Cross-Library Problem(s):

Here is an example of the complications a user may find while building applications using vtable verification in particular when there are dependencies on libraries that may or may not have been built with vtable verification.

Consider the following small test case:

lib.h:

---

```
extern "C" int printf(const char *, ...);

struct Base {
  virtual ~Base() { printf("In Base destructor\n"); }
};

Base * GetPrivate();
void Destroy(Base *);
```

---

lib.cc

---

```
#include "lib.h"

struct Derived_Private : public Base
{
  virtual ~Derived_Private()
  { printf("in Derived_Private destructor\n"); }
};

Base * GetPrivate()
{
  return new Derived_Private();
}

void Destroy(Base * pb)
{
  delete pb;    // virtual call #1
}
```

---

main.cc

---

```
#include "lib.h"
```

```
struct Derived: public Base
 {
  virtual ~Derived()
  { printf("In Derived destructor\n"); }
};

int main()
{
  Derived * d = new Derived;
  Destroy(d);
  Base * pp = GetPrivate();
  delete pp;   // Virtual call #2
}
```
---

- lib.h is the interface to lib. It just defines a simple class with a virtual method (in this case destructor)
- lib.cc privately declares its own derived class ("Derived_Private") from "Base" and provides an implementation of the virtual method
- main.cc also defines its own derived class from "Derived".
- Note that lib.cc and main.cc do not know anything about each other's derived classes and, therefore, the compiler generated vtables for each one.
- There are two virtual calls in this example: one within "Destroy" (virtual call #1) and one within “main” (virtual call #2).

User case scenarios:

1. Both library and main executable are build with vtable verification:

   g++ -fvtable-verify=std -fpic lib.cc -shared -o verified_lib.so
   g++ -fvtable-verify=std -fpic main.cc verified_lib.so

   This works correctly, the vtable verification runtime knows about both derived classes and corresponding vtables and has no problem verifying the pointers to vtables are correct.

2. Only the main executable is built with vtable verification. This scenario may be common since users may get libraries that they cannot build themselves.

   g++ -fpic lib.cc  -shared -o lib.so
   g++ -fvtable-verify=std -fpic main.cc lib.so

   This scenario fails with this example under the current implementation. In this case the virtual call #2 cannot be verified because the vtable verification runtime does not know anything about the vtable created for class "Derived_Private" since the library was not

build with vtable verification.

If you **must** use a library or plugin that was not compiled with verification, the best way to make it work, in a situation like this, is to write and link in your own version of __vtv_verify_fail, which is the function that the verifier calls when it cannot verify a vtable pointer. In your own version of __vtv_verify_fail you can check to see if the failure occurred in the non-verifiable library, and handle that case specially (including returning the vtable pointer as valid, if you like), and calling abort otherwise.

3. Only the library is built with vtable verification.

```
g++ -fvtable-verify=std -fpic lib.cc -shared -o verified_lib.so
g++ -fpic main.cc verified_lib.so
```

This scenario also fails with this example under the current implementation.
In this case, virtual call #1 cannot be verified because the vtable verification runtime does not know anything about the vtable created for class "Derived" since the main executable was not build with vtable verification.

The verification problem in this scenario only happens if the main executable extends the class hierarchy provided by the library (in the example “Derived” extends “Base”). If the user is not extending the library class hierarchy it is ok to use a verified library in a non-verified executable.