Vtable Verification Feature Proposal

- Problem Description
- Solution Constraints
- Proposed Solution
  - General Overview
  - More Details - Collecting the Data and Accessing the Data Structures
  - More Details - Building the Data Structures
  - More Details - Verifying the Virtual Method Dispatches
  - Security Analysis
  - Miscellaneous Implementation Notes
  - Other

Problem Description

We would like to improve the security of programs and 'harden' them against certain types of security attacks, where by 'harden' we mean 'make it much more difficult for attacks to succeed'. The particular types of attacks we are concerned with are attacks where the attacker manages to hijack control flow of a program, thereby allowing his or her own malicious code to be executed. There are several ways in which a hijacker can seize control of the program:

1). The attacker might corrupt the return address on the stack, so when a function call finishes executing, instead of returning to the call site, it returns either directly to the attacker's code, or to a function in one of the standard libraries that will allow the attacker to create a new shell and then execute his/her own code (the latter type of attack is known as a "return-to-libc" attack).

2). The attacker might overwrite the contents of a function pointer in the program, again making it go someplace other than its original target, and eventually allowing the attacker to gain control of the program. Programs written in C++ are particularly vulnerable to such attacks, as they contains many function pointers in their vtables.

A common type of attack is to exploit use-after-free (or similar) bugs in the code. In this case there are objects that get freed but to which there still exist valid pointers in the program, so the program will attempt to use these objects as if they are still valid. Once such an object has been freed, an attacker can cause the same bit of memory to be reallocated, and overwrite the vtable pointer in the object to point to the attacker's fake vtable. Then when the object is accessed by the program, it goes and executes the attacker's code.

The purpose of this document is to describe an approach whereby the compiler can harden programs against such attacks, allowing corruption of vtable pointers in objects to be detected before any attempt to call through the invalid pointer, and for execution to be halted if such a corruption is detected.

Solution Constraints

In our case there are several important constraints on any solution to this problem:

1. We want our solution to be both complete and precise. Every virtual method dispatch is a potential vector of attack so every virtual method dispatch must be checked. Furthermore, it is not sufficient to make sure that an object's vtable pointer points to just any valid vtable (or range of addresses within a set of valid vtables). If an object's vtable pointer points to a valid vtable for a different type of object (particularly for an object of a different size), attackers can still make use of this to wreak havoc. We need to be able to detect when a virtual call is through a vtable whose static type is inconsistent with the object's dynamic type.

2. The solution must be secure, i.e. it must not open up any new vectors for attack.

3. The solution must not change the C++ ABI. Otherwise the compiler changes will not be acceptable to the open source community that maintains GCC (our compiler).

4. The solution cannot use RTTI or exception handling internals, since some of the programs we are interested in protecting are built with -fno-exceptions -fno-rtti.

Proposed Solution

General Overview

For any given C++ object with virtual methods, the set of valid vtables required for the virtual methods can be computed based on inheritance relationship information that the compiler can collect at compile time. Therefore, we will have the compiler collect this information by building a set of mappings from the base classes to derived classes, and recording all the vtable pointers for all these classes. At run time these mappings will be compiled into data structures that indicate, for every potential polymorphic base class (where "polymorphic" means a class that needs a virtual table), the complete set of valid vtable pointers that an object declared to be of that base class might legally contain. At each virtual call site we will insert a call to a verification function before we dereference the vtable pointer for the virtual call. The verification function will take an indicator of the statically declared type of the object and the actual vtable pointer in the object. It will use the data structure previously mentioned to verify that the vtable pointer in the object is a valid vtable pointer for the declared type. If verification succeeds, then the virtual method dispatch is allowed to take place. If verification fails, execution is immediately halted. This will guarantee that for any virtual dispatch in the program, the vtable being used is for the virtual dispatch is

1). a valid vtable for the current program; and

2). a member of the set of valid vtables for the call site.

This solution can be divided roughly into three main pieces:

1. Collecting the class hierarchy and vtable pointer data.
2. Building verification data structures with the collected data.
3. Doing the actual verification of vtable pointers using the built data structures.

Each of these pieces is discussed in more detail below.

Note: This work is predicated on the fact that the vtables themselves are normally loaded into read-only memory, so attackers cannot modify the vtables directly. On the other hand, program objects get allocated (and deallocated) in read-write memory, so that is where the vulnerability is.

More Details - Collecting the Data and Accessing the Data Structures

The compiler, while parsing classes, builds up a graph of class hierarchies, for those classes that have vtables (or any of their descendant classes), and also records all the vtable pointers for those classes.

For each class in the graph, the compiler generates a global pointer variable, which we refer to as the vtable map variable for that class. The vtable map variable for a class will, at run time, be set to point to the data structure containing the set of valid vtable pointers for that class. When the compiler initially creates these variables, they are set to NULL.

Since these vtable map variables will be used by the verification process to verify that the vtable pointers in objects are valid, it is important that the vtable map variables themselves be safe and not vulnerable to being overwritten by attackers. For this purpose, when the compiler creates the vtable map variables, it puts them all in a special named section in the object file. Once the vtable map variables are set to point to the data structures (which happens very early in program start up, before 'main'), the section containing the vtable map variables is found and set to be read-only (via a call to 'mprotect'). Unless a dlopen necessitates a vtable map variable being updated (explained later in this document), vtable map variables remain read-only for the rest of the program execution.

Since multiple source files (and object files) may see the same class definitions, there may be multiple definitions of any particular vtable map variable. For that reason, vtable map variables are created as COMDAT variables, which tells the linker that it is ok for there to be multiple definitions of the same variable, and the linker just picks one and uses that. If there is more than one active version of a vtable map variable for a particular class (which can happen if some vtable map variables are not truly global), then we make sure that all the vtable map variables for a particular class point to the same data set (only one data set per class).

More Details - Building the Data Structures

In order for the compiler to cause the data structures to be built at run time (rather than compile time), which is a requirement because the actual values of the vtable pointers won't be known until run time, we use two mechanisms. First, we wrote a library function (to go into libvtv, a new runtime library), to which we can pass a vtable map variable for a class and a valid vtable pointer for the class. The library function, __VLTRegisterPair, then does two things. First it checks to see if the vtable map variable is still NULL, in which case it creates an empty hash table and sets the vtable map variable to point to it. Then it puts the vtable pointer into the hash table. Below is a rough idea of the function:

```
void
__VLTRegisterPair (void **vtable_map_var, void *vtbl_ptr)
{
        if (*vtable_map_var == NULL)
           *vtable_map_var = init_hash_table ();
        hash_table_insert (*vtable_map_var, vtbl_ptr);
}
```

For purposes of efficiency, we have also written another function, __VLTRegisterSet (also in libvtv), which is a variation on __VLTRegisterPair, and which takes an array of vtable pointers and puts them all into the set for a particular class, allowing us to reduce the number of function calls we have to make to build up the data sets.

The second mechanism we use are constructor init functions. These are functions that the compiler inserts into each object file (in C++), in order to have a place to call any constructors that need to be called before ‘main’ begins execution. In each object file, we add a constructor init function for constructing our data structures. For each class hierarchy graph that the compiler built during the data collection phase, it traverses the graph and inserts a call to __VLTRegisterPair (or __VLTRegisterSet) into the constructor init function, passing in the vtable map variable for the class whose set we are building, and the vtable pointer (or array of pointers) to be inserted in the set of valid pointers for that class.

In order to make sure that the data structure itself is safe from attack we first wrote our own memory allocation scheme (based on mmap) so we can keep track of which pages are used for our data structures. We also create two special constructor init functions, one to make our vtable map variables and mmap’ed pages read-write, and one to make them read-only (via calls to __VLTChangePermission, in libvtv, which uses calls to ‘mprotect’). We use initialization priorities to control the order in which these constructor initialization functions are called: the one that makes our data read-write has a priority of 98; our constructor initialization functions that call __VLTRegisterPair and __VLTRegisterSet all have priorities of 99; and the one that makes our data read-only has a priority of 100. (The lower the priority, the earlier the function executes.)

In some very special cases, it may be necessary for our constructor init functions to run earlier than usual (i.e. before the dynamically loaded libraries get initialized). In that case we put them into the preinit array (by compiling with the flag “-fvtable-verify=preinit”). In this case we have two special preinit functions for setting the protections appropriately on our data. We make sure the function that makes our data read-write goes into the preinit array before any of our regular functions; and the function that makes our data read-only goes in after (see ‘Miscellaneous Implementation Notes’ for more details).

More Details - Verifying the Virtual Method Dispatches

In order to verify vtable pointers before they are used for virtual method dispatches, we have written a third library function, __VLTVerifyVtablePointer (again, in libvtv). This function takes two arguments, a vtable map variable for the statically declared type of the object, and the vtable pointer that is in the object. It then looks up the vtable pointer in the data structure pointed to by the vtable map variable (which is supposed to contain the set of all valid vtable pointers for the class or any of its descendants). If the vtable pointer is found, then it is valid and execution continues; otherwise execution halts. The code for __VLTVerifyVtablePointer looks roughly like this:

```
void *
__VLTVerifyVtablePointer (void **vtable_map_var, void *vtbl_ptr)
{
         if (hash_table_find (vtable_map_var, vtbl_ptr))
            return vtbl_ptr;
         else
           {
              __vtv_verify_fail (vtable_map_var, vtbl_ptr);   // By default this calls 'abort'
             return vtbl_ptr;                                // Normally this line will not be reached
           }
}
```

During the compilation process, when the compiler encounters a virtual method dispatch it first identifies the declared type of the object through which the call is being made. Then it finds the vtable map variable for that type. Finally it inserts a call to __VLTVerifyVtablePointer before the virtual method dispatch, and passes in the vtable map variable, and the vptr field from the object. At runtime, the verification call occurs before the virtual dispatch, and if the vtable pointer has been corrupted the verification will call __vtv_verify_fail, which normally prints an error message and terminates execution.

Security Analysis

The vtables themselves are placed by the compiler into read-only memory. They remain there for the entire execution of the program. Therefore the vtables (and the function pointers inside them) cannot be overwritten by hackers and are “safe”.

In order to hijack our verification procedure, a hacker would need to corrupt either a vtable map variable, making it point to some bogus data structure containing bogus vtable pointers; or he would need to corrupt one of our data structures directly, inserting bogus vtable pointers into the data structure.

The data structures that contain the sets of valid vtable pointers for each potential base class are allocated via our memory allocation functions (which use mmap). We keep track of which pages those are. We use mprotect to make the pages read-write before we need to update the data structures, and again to make them read-only after we are done updating them. These pages are therefore vulnerable to hackers only when we are updating the data structures ourselves. This will be during initial set up, before the function ‘main’ is reached; and during any call to dlopen, when we are loading a dynamic library (written in C++ and compiled with verification) which needs to update the data structure. It seems highly unlikely that a hacker will be able attack the program before it has even reached ‘main’, so we are not worried about that at this time. It is possible that a hacker could possibly cause a dlopen to happen on one thread and try to simultaneously overwrite/corrupt our data structure on another thread.

The vtable map variables, which point to the data structures to be used for verification, are placed by the compiler in read-only memory. When we make the data structures writeable, we also find the read-only section where the vtable map variables are placed, and make that writeable as well, to allow the variables to be updated to point to the correct data structures. When we make the data structures read-only, we make the vtable map variables read-only as well. Therefore, as with our main data structure, the vtable map variables are possibly vulnerable during a dlopen.

The last bit of data that we protect are a few static book-keeping variables we use in our memory allocation scheme.   We do not want an attacker to be able to change our protected page data. Therefore we put these variables in the same section into which we put the vtable map variables, and they are protected in the same way.

Miscellaneous Implementation Notes

This whole mechanism is controlled by a compiler flag, “-fvtable-verify”. This flag actually comes it three versions “-fvtable-verify=std”, “-fvtable-verify=preinit” and “-fvtable-verify=none”. “-fvtable-verify=none” is used to explicitly turn off vtable verification during the compilation process (but will not affect already compiled libraries and object files). The values ‘std’ and ‘preinit’ control when the constructor initialization functions for building our verification data structures get executed.   ‘std’ causes them to get executed in the normal execution order, i.e. after the .so files have been loaded and initialized (but still before ‘main’). ‘preinit’ causes our constructor initialization functions to be put into the special .preinit_array, which causes them to execute before the .so files get initialized.

The ‘preinit’ option was found to be necessary because of uses of tcmalloc. This is a C++ allocator with virtual dispatch. Therefore calls to malloc that get converted to calls to tcmalloc may end up being verified via __VLTVerifyVtablePointer. However such verifications will fail unless the data structure for tcmalloc has been built before the verification happens. Unfortunately, calls to malloc are replaced by tcmalloc invisibly (as far as the compiler is concerned), i.e. the compiler has no idea tcmalloc is involved, so it doesn’t know that it needs to construct tcmalloc’s data structures there. This means that if any .so file has an initializer that calls malloc (and some of them do), the only way to ensure the verification succeeds is to make sure tcmalloc’s constructor initialization functions have run first (by putting them into the .preinit array).

Any program that is compiled with -fvtable-verify=... needs to be *linked*, through the gcc/g++ driver, with -fvtable-verify=... as well. This is because the compiler driver adds a few critical things to the link command, without which this feature will not work properly:

In order to be able to use 'mprotect' to change the permissions on our data and *only* on our data, we have to make sure that our data is on pages that are not shared with any other data ('mprotect' works only at the page granularity). To do this, we create our own named section ('.vtable_map_vars'), and make the section page-aligned. We also pad the end of the section with a page-sized amount of zeros. Then we put all of our vtable map variables and bookkeeping data into this section. To make the section page-aligned, we make sure that the very first thing ever to go into the section is a page-aligned variable, named 'vtable_map_vars_start'. To pad the section, we make sure that the very last thing to ever go into the section is a variable that contains a page-sized array of zeros, named 'vtable_map_vars_end'. 'vtable_map_vars_start' and the constructor init function that makes our data read-write are defined in vtv_start.o; 'vtable_map_vars_end' and the constructor init function that makes our data read-only are defined in 'vtv_end.o'. When the gcc driver sees '-fvtable-verify=std', it adds vtv_start.o to the link line just after crtbegin.o, and it adds vtv_end.o to the link line just before crtend.o. This ensures that 'vtable_map_vars_start' is the first thing to go into the .vtable_map_vars section, and 'vtable_map_vars_end' is the last thing. It also adds '-u_vtable_map_vars_start -u_vtable_map_vars_end' to the link line, to prevent the linker garbage collector from eliminating those two variables as garbage.

We also have vtv_start_preinit.o and vtv_end_preinit.o, which are similar to vtv_start.o and vtv_end.o, and are substituted for them if -fvtable-verify=preinit is used rather than -fvtable-verify=std.

As mentioned previously, to ensure that our constructor initialization functions run before any 'regular' constructor initialization functions when 'std' is used, we take advantage of initialization priorities. Users are allowed to use priorities 101 through 65535. Priorities 1 through 100 are reserved for the system (including the compiler). We give our constructor initialization functions the priority 99. The function that makes our data structures read-write has priority 98, and the one that makes our data structures read-only has the priority 100.

Other

Calls to dlopen. Any dynamically loaded library that was built with verification will have its own constructor initialization function that will contain calls to update the vtable map data structures. This function will be executed as part of loading/initializing the library. As usual, there will be a call to __VLTChangePermission to make our data writeable before the main constructor init function is called, and another call afterwards to make our data readonly again.

There can be problems if there are libraries that are built without verification, and the main program is built with verification (or vice versa), *and* something on one side of this verification boundary extends a class defined on the other side of the boundary, *and* passes objects across the boundary.   The best solution, in this case, is for the programmer to write and link in an alternative version of the function __vtv_verify_fail (which gets called from __VLTVerifyVtablePointer when regular verification fails) that can handle these interoperability failures in whatever manner seems best to the programmer (ignore them, do some additional checking, or whatever). __vtv_verify_fail is written in such a manner as to allow a programmer to statically link in a replacement version.

The signature for __vtv_verify_fail is

```
void __vtv_verify_fail (void **vtable_map_var, const void *vtbl_pointer);
```

The arguments that get passed in to this function are the same as those that were passed to __VLTVerifyVtablePointer, i.e. the pointer to the data structure for the statically declared type of the object, and the vtable pointer from the object that needs to be verified. If execution returns from __vtv_verify_fail, then __VLTVerifyVtablePointer assumes that the secondary verification succeeded, and returns vtbl_pointer to the call site, for use in the virtual method dispatch.

libstdc++ itself will normally be built with vtable verification. However it is also built containing stub (do-nothing) versions of the functions __VLTVerifyVtablePointer, __VLTRegisterPair, __VLTRegisterSet, and __VLTChangePermission. So vtable verification will not be turned on with libstdc++ unless the program is also explicitly linked with libvtv, which is the runtime library that contains the “real” versions of these functions.