Improving Function Pointer Security for Virtual Method Dispatches

GNU Tools Cauldron Workshop 2012

Caroline Tice
cmtice@google.com

Talk Overview

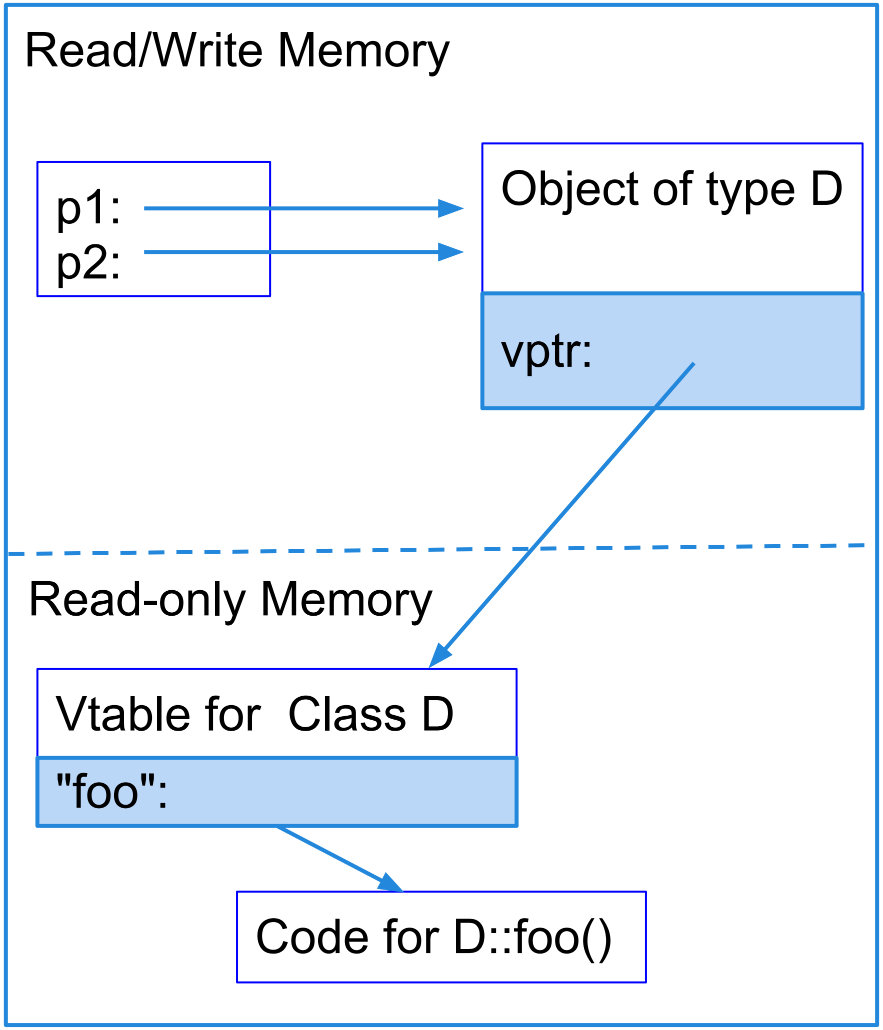
- Motivation - The Problem
- Our Approach - Overview
- Some Gory Details
- Problems & Status
- Discussion

Motivation - The Problem

- Hackers!
- "The Cloud" platform.
- Browsers == ~~Targets.~~ BIG Targets!
- Chrome (C++).
  - security
  - speed
- "80% attacks exploit use-after-free..."
  - Use bug to gain control of function pointers.

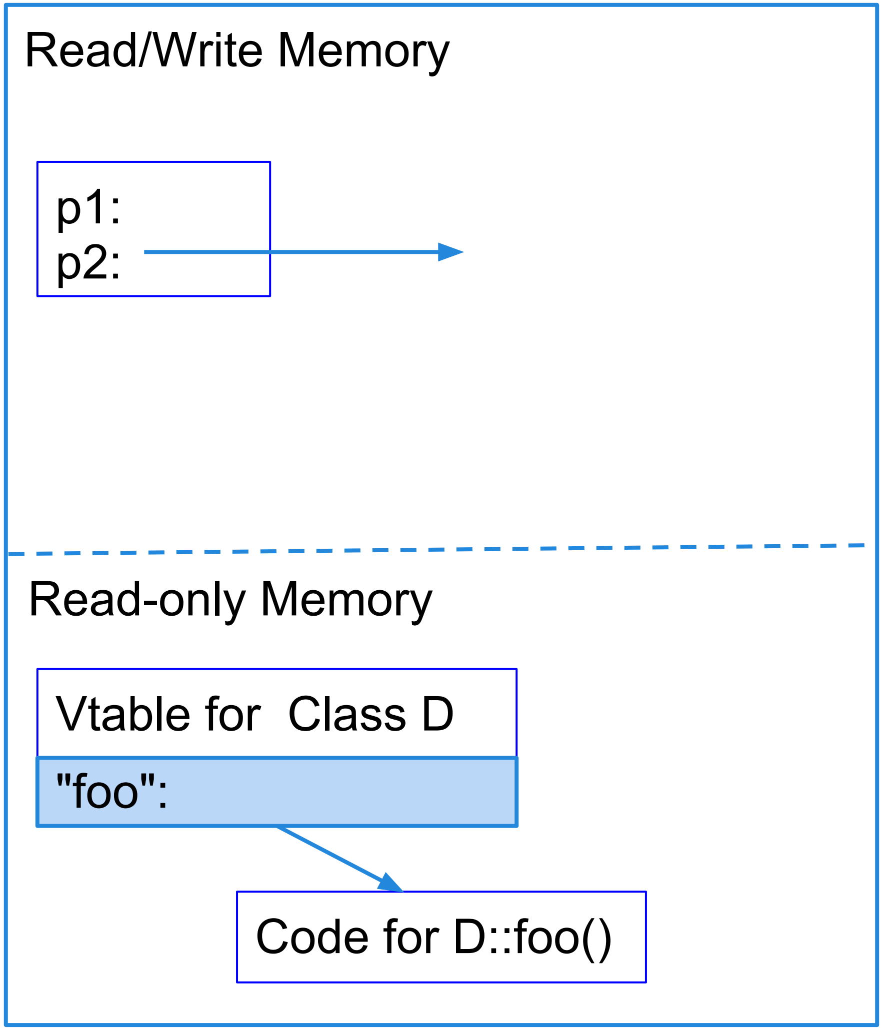
Use-after-free Bug

```
class B {
public:
    int virtual foo ()
{...}
};

class D : public B {
public:
    int virtual foo ()
{...}
};
...
D *p1 = new D();
D *p2 = p1;   // alias
p1-> foo ();  // 1st use
...
delete (p1);  //"free"
...
p2->foo (); // BAD use!
```



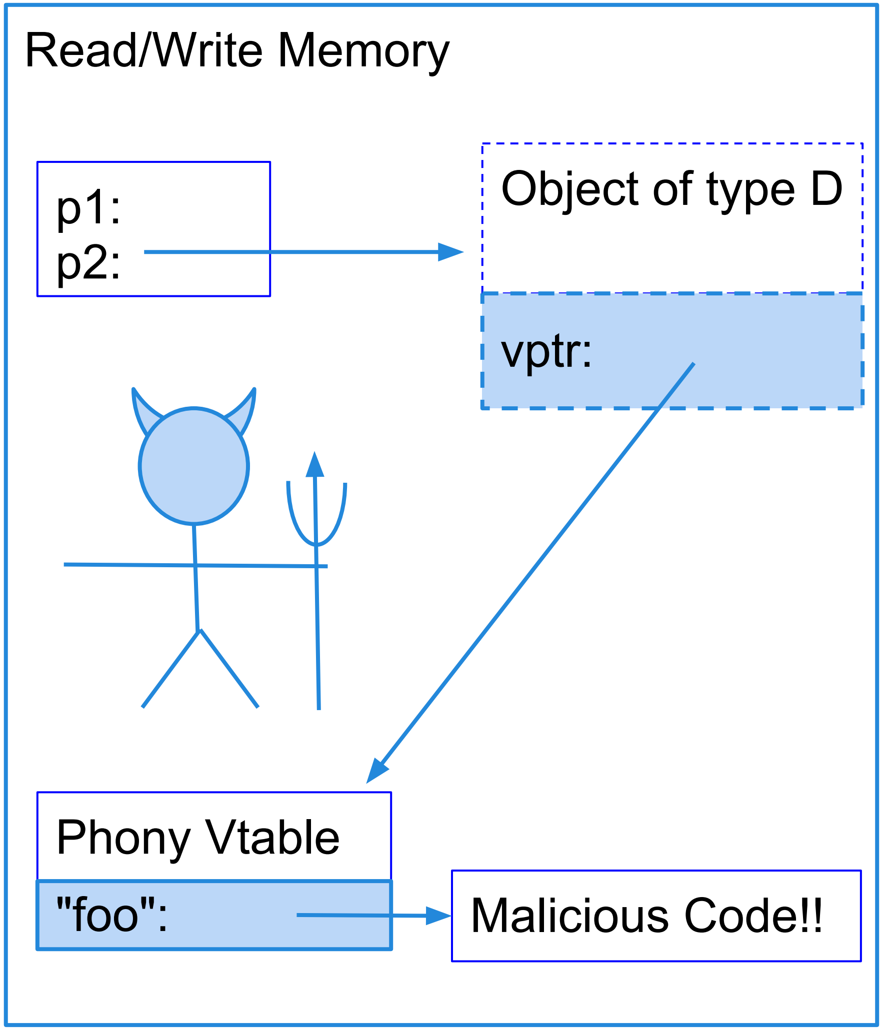
Use-after-free Exploit

```
class B {
public:
    int virtual foo ()
{...}
};

class D : public B {
public:
    int virtual foo ()
{...}
};
...
D *p1 = new D();
D *p2 = p1;   // alias
p1-> foo ();  // 1st use
...
delete (p1);  //"free"
...
p2->foo ();  // BAD use!
```



Use-after-free Exploit

```
class B {
public:
    int virtual foo ()
{...}
};

class D : public B {
public:
    int virtual foo ()
{...}
};
...
D *p1 = new D();
D *p2 = p1;   // alias
p1-> foo ();  // 1st use
...
delete (p1);  //"free"
...
p2->foo ();  // BAD use!
```



Desired Solution Constraints

- Complete and Precise.
- Introduce no new attack vectors.
- No more than 10% performance loss.
- No C++ ABI changes.
- No RTTI.

Our Approach - Overview

- Modify GCC to collect information.
  - Class hierarchies.
  - ALL vtable pointers.
  - Pass info to library routines.
- Make GCC modify virtual call sites.
  - Insert verification call before virtual method dispatch.

Our Approach - Overview (cont.)

- New Library Routines (in libsupc++).
  - Build data structure from collected data.
  - Put data structure in protected memory.
  - Verify vtable pointer is in set of valid pointers for a given base class.
  - Update data structure on dlopen.

Our Approach - Alternate Explanation

```
class B {
public:
    virtual int foo ()
{...}
};

class D : public B {
public:
    virtual int foo ()
{...}
};

B *b_ptr;
D d_obj;
b_ptr = &d_obj;

b_ptr-> foo ();
```

```
D.1 = b_ptr;
D.2 = b_ptr->_vptr.B;



D.3 = *D.2;
D.4 = call(D3 + offset)(D.1);
```

Our Approach - Alternate Explanation

```
class B {
public:
    virtual int foo ()
{...}
};

class D : public B {
public:
    virtual int foo ()
{...}
};

B *b_ptr;
D d_obj;
b_ptr = &d_obj;

b_ptr-> foo ();
```

```
D.1 = b_ptr;
D.2 = b_ptr->_vptr.B;
D.5 = "class B";

D.6 = VerifyVtablePointer (D.5, D.2);
D.3 = *D.6;
D.4 = call(D3 + offset)(D.1);
```

Our Approach - Alternate Explanation

```
class B {
public:
    virtual int foo ()
{...}
};

class D : public B {
public:
    virtual int foo ()
{...}
};

B *b_ptr;
D d_obj;
b_ptr = &d_obj;

b_ptr-> foo ();
```

```
D.1 = b_ptr;
D.2 = b_ptr->_vptr.B;
D.5 = "set of valid vtable pointers
for class B";
D.6 = VerifyVtablePointer (D.5, D.2);
D.3 = *D.6;
D.4 = call(D3 + offset)(D.1);
```

Our Approach - Alternate Explanation

```
class B {
public:
    virtual int foo ()
{...}
};

class D : public B {
public:
    virtual int foo ()
{...}
};

B *b_ptr;
D d_obj;
b_ptr = &d_obj;

b_ptr-> foo ();
```

```
D.1 = b_ptr;
D.2 = b_ptr->_vptr.B;
D.5 = &_"set of valid vtable pointers
for class B";
D.6 = VerifyVtablePointer (D.5, D.2);
D.3 = *D.6;
D.4 = call(D3 + offset)(D.1);
```

Vtable Verification Function

```
void *
VerifyVtablePointer (set *valid_vtbl_ptrs, void *vtbl_ptr)
{
    if (member (vtbl_ptr, valid_vtbl_ptrs))
        return vtbl_ptr;
    else
        abort ();
}
```

Main Questions...

"Set of valid vtable pointers for class B"

- How do we build it?
- How do we reference it?
- How do we protect it?

```
D.1 = b_ptr;
D.2 = b_ptr->_vptr.B;
D.5 = & "set of valid vtable pointers for
class B";
D.6 = VerifyVtablePointer (D.5, D.2);
D.3 = *D.6;
D.4 = call(D3 + offset)(D.1);
```

How to Build Set of Valid Vtable Pointers

- Part 1. Collect the Data.
  - Done at compile time.
  - Data stored in object file.
- Part 2. Build Searchable Data Structure.
  - Done at run time.
  - Linker/loader fills in pointer values.
  - Question: Which tool builds data structure?
    - Answer: Compiler (for now).

How Does Compiler Build Data Structure at Run Time?

By using two facilities:

1. Constructor initialization functions.
    a. Standard part of C++.
    b. Run between "_start" & "main".
    c. Used to initialize things (objects) needed by main.
2. New library function.
    a. Add function to build data structure to C++ library.
    b. Call function from constructor init function.
    c. Pass function data compiler collects.

Example of Our Constructor Initialization Function

Standard tag to mark constructor init functions

Initialization priority

Unique part of function name, based on filename

New library functions

```
_GLOBAL__sub_I.00099.my-file ()
{
    ChangePermission ("rw");
    RegisterPair (class, vtbl_ptr);
    RegisterPair (class, vtbl_ptr);
    ...
    RegisterPair (class, vtbl_ptr);
    ChangePermission ("ro");
}
```

Inserting the Verification Calls

New tree pass ("vtable verify")!

- Controlled by new flag (-fvtable-verify).
- Just before converting gimple to RTL.
- Finds and modifies all virtual calls.

Main Questions...

"Set of valid vtable pointers for class B"

- How do we build it?
- *<u>How do we reference it?</u>*
- How do we protect it?

```
D.1 = b_ptr;
D.2 = b_ptr->_vptr.B;
D.5 = & "set of valid vtable pointers for
class B";
D.6 = VerifyVtablePointer (D.5, D.2);
D.3 = *D.6;
D.4 = call(D3 + offset)(D.1);
```

How Do We Reference the Data Structure?

We introduce vtable map variables!

```
D.1 = b;
D.2 = b->_vptr.B;
D.5 = _ZTV1B.vtable_map;
D.6 = call VerifyVtablePointer (D.5, D.2);
D.3 = *D.6;
D.4 = call (D.3 + offset) (D.1);
```

What ARE vtable map variables?

- Global comdat vars created by compiler.
- Initialized by compiler to NULL.
- Really initialized by RegisterPair.
  - Used as first argument to RegisterPair.

    "RegisterPair (vtbl_map_var, vtbl_ptr);"
- All placed in same named section.
- Each has unique comdat name.

Functions to Build & Use Data Structures

```
void
RegisterPair (void **vtbl_map_var, void *vtbl_ptr) {
    if (*vtbl_map_var == NULL)
        *vtbl_map_var = new_hash_table ();
    hash_table_insert (*vtbl_map_var, vtbl_ptr);
}

void *
VerifyVtablePointer (void **vtbl_map_var, void *vtbl_ptr)
{
    if (hash_find (vtbl_map_var, vtbl_ptr))
        return vtbl_ptr;
    else
        abort ();
}
```

Problem with Vtable Map Variables (and Data Structure).

- Potential security hole.
  - Must use our own memory allocation
    - Used for allocating new data structure(s).
  - Must be mprotect'ed by ChangePermission.
    - Called at start & end of init function.
    - Find section containing vtable map vars & set protections on it, too.
  - Must also protect memory protection bookkeeping data.

Three New Library Functions

In libsupc++ (part of libstdc++):

- ChangePermission
  - Un-protects & re-protects memory for us.
- RegisterPair
  - Initializes data structures & vtable_map vars.
  - Adds vtable pointers to class data structures.
- VerifyVtablePointer
  - Looks for vtable pointer in class data structure.
  - Aborts if not found.

Putting It All Together...

```
Class B {
public:
    virtual int foo ();
};

Class C: public B {
public:
    virtual int foo ();
};

Class D : public C {
public:
    virtual int foo ();
};
```

```
B *b_obj;
D d;

b_obj = &d;
```

[VerifyVtablePointer (_ZTV1B.vtable_map, b_obj.vptr &_ZTV1D)]

```
b_obj->foo (); /* virtual call */
```

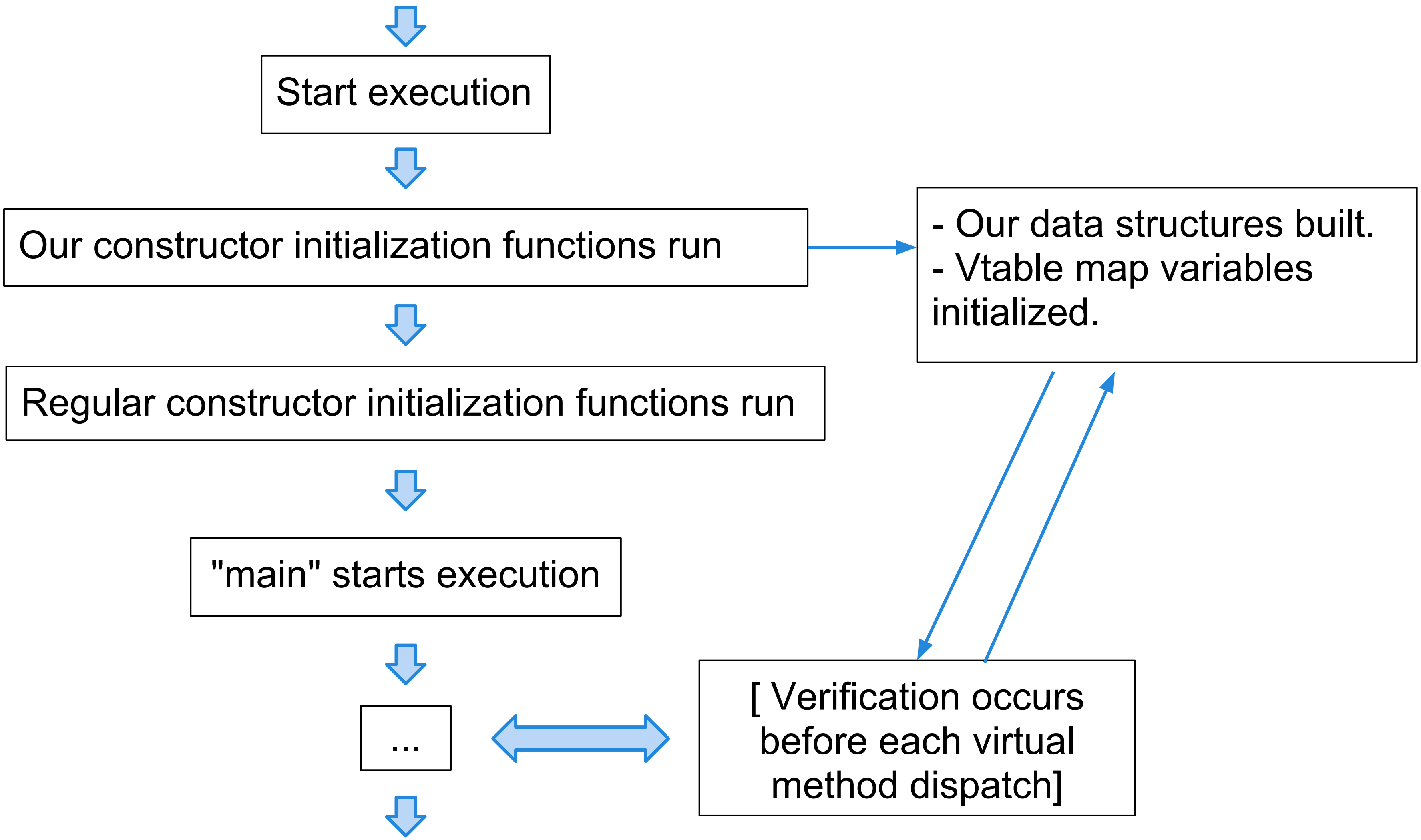
Data structure(s) contents:

_ZTV1B.vtable_map → &_ZTV1B, &_ZTV1C, &_ZTV1D

_ZTV1C.vtable_map → &_ZTV1C, &_ZTV1D

_ZTV1D.vtable_map → &_ZTV1D

Execution Model...



"tcmalloc" - an ugly problem...

- tcmalloc replaces malloc *(invisibly!!).*
- tcmalloc written in C++.
- tcmalloc uses virtual function calls.
  - Calls to "malloc" get verified.
  - Some calls to malloc can occur in .so files.
  - .so files are loaded & initialized *before constructor initialization functions are executed*.

==> Calls to tcmalloc with empty data structures failed to verify!!

"tcmalloc" Solution...

- Don't use tcmalloc.
- Don't write it in C++.
- REALLY don't use virtual method calls in it...

Wishful thinking! Not an option...

Real "tcmalloc" Solution...

- Need to force tcmalloc's init functions to run before .so's are initialized...
  - Put them into the .preinit array!
  - Control via flag.
    - "-fvtable-verify=std"
    - "-fvtable-verify=preinit"

Mixing/Matching Protections Across Library Boundaries...

Library

public interface:

```
class B;
B* B::GetPrivate ();
virtual B::~B();
```

private implementation:

```
class P : public B {...};
virtual P::~P();

B* B::GetPrivate (){
    return new P();
}
void Destroy (B *b) {
    delete pb; // vcall 1
}
```

User Program

```
class D : public B {...};
virtual D::~D() {...};

int main () {
    ...
    D * d = new D();
    Destroy (d);
    B * pp = B::GetPrivate();
    delete pp; // vcall 2
    ...
}
```

Mixing/Matching Protections Across Library Boundaries...

- Possible solutions:
  - Option to disable verification on classes defined in certain .h files or directories.
  - For unprotected library, fall back on secondary verification:
    - Collect vtable symbols & pointers for library, on load.
    - Search if vtable-map var is for one of vtable symbols, search for vptr value in pointer list.
  - For unprotected main:
    - Generate stub/do-nothing versions of library functions.
    - Two versions of stdlibc++, one with & one without verification.

Other Difficulties Encountered...

- Hard to find all the vtable pointers.
  - instantiated templates
  - construction vtables
- Had to write our own memory allocation.
- Too many calls to ChangePermissions.
- Constructor initialization ordering.

Current Status

- Done: (First) prototype implementation.
  - Data collected.
  - Constructor initialization functions generated.
  - Verification calls inserted.
  - Library functions written.
  - Memory allocation & protection written.
- To do:
  - Detailed performance measuring & tuning.
  - (Possibly) revisit some design decisions.
  - Submit patches to GCC trunk.

Acknowledgements (Blame list?)

Co-implementor:

- Luis Lozano

Co-designers/Consultants:

- Ken Buchanan
- Cary Coutant
- Lawrence Crowl
- Ulfar Erlingsson
- Bhaskar Janakiraman
- David Li
- Diego Novillo
- Paul Pluzhnikov
- Geoff Pike
- Sriraman Tallam

Questions/Discussion

- Comments?
- Suggestions?
- Potential discussion topics:
  - General approach.
  - "tcmalloc" problem/solution.
  - Mix & match verification across library boundaries.

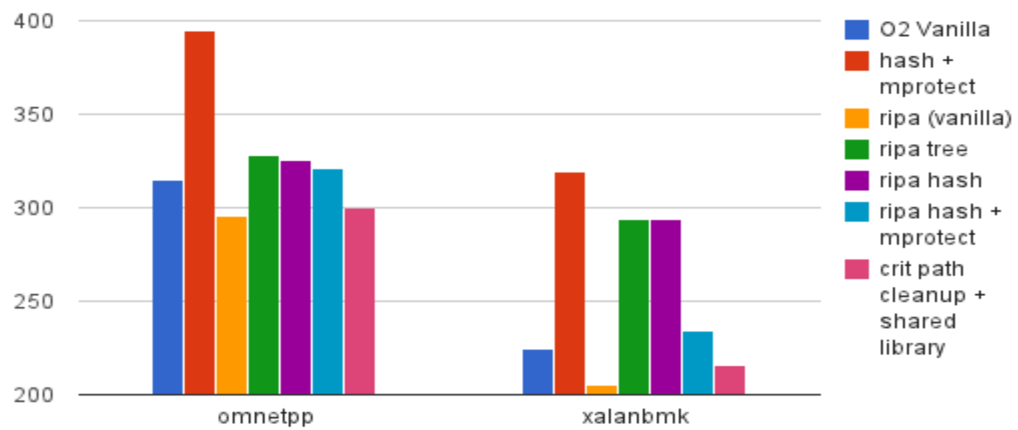
svn://gcc.gnu.org/svn/gcc/branches/google/mobile-4_6-branch/vtable-security

Back-up Slides start here...

Performance?

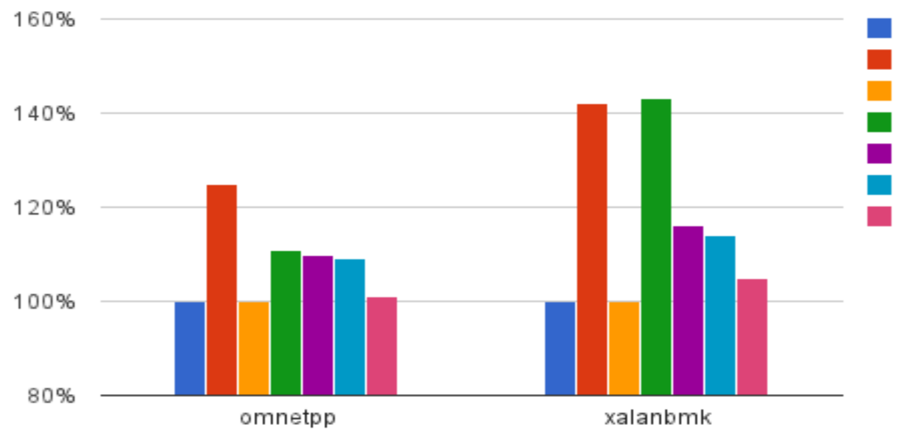
- Cost to insert dummy verification calls: 5-10%
- ChangePermissions
  - per object file: 400-700% slowdown!!
  - per binary: ~350ms (still noticeable)
- [Hash table wastes lots of space]

SPEC Performance Numbers



	vanilla	hash mprotect	ripa (vanilla)	ripa tree	ripa hash	ripa hash + mprotect	crit path cleanup & shared lib
omnetpp	315	395	295.84	327.85	325.32	321.36	299.74
xalanbmk	225	319	205.13	294.13	294.09	234.29	216.1

SPEC Performance Numbers (cont.)



	vanilla	hash + mprotect	ripa (vanilla)	ripa tree	ripa hash	ripa hash + mprotect	crit path cleanup & shared lib
omnetpp	100%	125%	100%	111%	110%	109%	101%
xalanbmk	100%	142%	100%	143%	116%	114%	105%

Existing Security Options

- Detecting programmer errors.
  - gcc: -D_FORTIFY_SOURCE=2, mudflap, ASAN, TSAN
  - external: Purify, Valgrind
- Preventing/discouraging attacks.
  - -fpie/-fpic, -Wformat options
- Detecting attacks.
  - -fstack-protector options

Collecting the Data - GCC Front End

Input
C++
Program

PARSING:
IFF class has vtable:
- Record base class / derived class info
- Record all vtables associated with each class

call cgraph_finalize_compilation_unit

class
hierarchy &
vtable data

- Generate OUR constructor initialization function
- Add function into the cgraph

How Verification Actually Works

For *each "base" class*:

- Collect set of all valid vtable pointers for that class or any of its descendant classes.
- Create "vtable map" variable to point to set of valid vtable pointers for the base class.
- VerifyVtablePointer takes two arguments
  - vtable map variable for *declared* (static) class of object
  - vptr value from object
- Look for vptr value in set pointed to by vtable map variable.

Issues and Open Questions

- "tcmalloc issue" (ordering problems).
- Extending classes across library boundaries.
- General approach.