SafeDispatch
Securing C++ Virtual Function Calls from Memory Corruption Attacks

Dongseok Jang  Zachary Tatlock  Sorin Lerner
UC San Diego  University of Washington  UC San Diego
Vulnerable
Control Flow Hijacking

Lead Program to Jump to Unexpected Code
That does what attacker wants

Example: Stack Buffer Overflow Attacks
Well studied and hard to be critical by itself

New Frontier: Vtable Hijacking
class C {
    virtual int foo();
    virtual int bar();
    int fld;
};
...
C *x = new C();
Virtual Call: 2-Step Dereferencing for Callee

```c
x->foo();

vptr = *((FPTR**)x);
f = *(vptr + 0);
f(x);
```

- **heap obj**
- **vtable**
Vtable Hijacking

\[ x \rightarrow \text{foo}(); \]

\[ \text{vptr} = *((\text{FPTR}**)x); \]
\[ f = * (\text{vptr} + 0); \]
\[ f(x); \]

heap obj  
vtable  

fake vtable

Arbitrary Code

foo's impl

bar's impl
C *x = new C();
x->foo();
delete x;
// forget x = NULL;
...

D *y = new D();
y->buf[0] = input();
...

x->foo();
Vtable Hijacking: Real Case

Vtable Hijacking of Chrome via Use-after-Free Pinkie Pie’s demonstration at Pwn2Own
Used to trigger ROP for sandbox escaping of Chrome

Found in IE, Firefox, Chrome
How to Prevent Vtable Hijacking?

With Accuracy & Low Overhead?
C *x = ...

Check(x);

x->foo();
C *x = ...  
`ASSERT(VPTR(x) ∈ Valid(C));`

```
C = { vptr of C or C’s subclasses }
```

Obtained by class hierarchy analysis (CHA)
\[ C \ *x = \ldots \]
\[ \text{ASSERT (VPTR}(x) \ \subseteq \ \text{Valid}(C)); \]
\[ x->\text{foo}(); \]

Simple Implementation Can Be Slow

Involved data structure lookup/function calls
C *x = ...  

\textit{ASSERT}(\text{VPTR}(x) \in \text{Valid}(C));

x->foo();  
vptr = *((FPTR**)x);  
f = *(vptr + 0);  
f(x);
Inlining Optimization

```c
C *x = ...

// ASSERT(VPTR(x) ∈ Valid(C));
vptr = *((FPTR**)x);
ASSERT(vptr ∈ Valid(C));
f = *(vptr + 0);
f(x);
```
C *x = ...

// ASSERT(VPTR(x) ∈ Valid(C));

vptr = *((FPTR**)x);

// ASSERT(vptr ∈ Valid(C));

ASSERT(vptr ∈ \{C::vptr, D::vptr\});

f = *(vptr + 0);

f(x);

Say that C has only one subclass D
→ Specialization of Checks
C *x = ...

// ASSERT(VPTR(x) ∈ Valid(C));

vpotr = *((F PTR**)x);

// ASSERT(vptr ∈ Valid(C));

// ASSERT(vptr ∈ {C::vptr, D::vptr});

f = *(vptr + 0);

f(x);

SAFE:
Inlining Optimization

```c
C *x = ...

// ASSERT(VPTR(x) ∈ Valid(C));

vpotr = *((FPTR**)x);

// ASSERT(vptrr ∈ Valid(C));
// ASSERT(vptr ∈ {C::vptr, D::vptr});

if (vpotr == C::vptr) goto SAFE;
if (vpotr == D::vptr) goto SAFE;
exit(-1);

SAFE: f = *(vpotr + 0);
f(x);
```
Inlining Optimization

How to Order Inlined Checked? → Profile-guided Inlining

```c
if (vptr == C::vptr) goto SAFE;
if (vptr == D::vptr) goto SAFE;
exit(-1);

SAFE: f = *(vptr + 0);
f(x);
```
C *x = ...

vptr = *(*(FPTR**)x);

ASSERT(vptr ∈ {C::vptr, D::vptr});

f = *(vptr + 0);

f(x)
C *x = ...  

vptr = *((FPTR**)x);

// ASSERT(vptr ∈ {C::vptr, D::vptr});

f = *(vptr + 0);

ASSERT(f ∈ ValidM(C,foo));

f(x)

Checking Callee Before It Is Called

Provides same security as vtable checking
Method Pointer Checking

C *x = ...

vptr = *(((FPTR**)x);

// ASSERT(vptr ∈ {C::vptr, D::vptr});

f = *(vptr + 0);

// ASSERT(f ∈ ValidM(C,foo));

ASSERT(f ∈ {C::foo});

f(x)

Save Checks for Shared Methods
A *x = ...
// m: index into a vtable
// x->*m can be any methods of A
(x->*m) ()

Say that A has 1000 methods
Vtable Checking Can Be Faster
Method Pointer Checking

Fewer Checks for Usual Virtual Calls

More Checks for Member Pointer Calls
Hybrid Checking

Method Checking for Usual Virtual Calls

Vtable Checking for Member Pointer Calls
Tamper Resistance

Inserted Checks in Read-Only Memory

Checking Data in Read-Only Memory
Chromium Browser
Realistic: ≈ 3 millions of C++/C LOC
Popular target of vtable hijacking

Running On JS, HTML5 Benchmark
Performance

Unoptimized (Avg: 23%)
Performance

Profile-Guided Inlining (Avg: 6%)
Performance

Inlined Method Ptr Checking (3%)
Performance

Hybrid Checking (Avg: 2%)
7% Code Size Increase

8.3 MB out of 119 MB

Checking Data + Inlined Checks
Future Work

Separate Compilation
Link-time CHA / inlining

Dynamic Link Library
Runtime update of checking data
Summary

Vtable Hijacking
Often happening in web browsers

Compiler-based Approach
Code Instrumentation / static Analysis

Realistic Overhead
Careful compiler optimizations
Thank you!

http://goto.ucsd.edu/safedispatch