Static detection of C++ vtable escape vulnerabilities in binary code

David Dewey      Jonathon Giffin
School of Computer Science
Georgia Institute of Technology
{ddewey, giffin}@gatech.edu
Common problem in C++

In C++ specifically, how does one convert an instance of an object into an instance of another object?

“…use static_cast in all cases and see what the compiler says.”

C++ Type confusion vulnerabilities

Adobe Flash Player SharedObject Type Confusion Vulnerability
CVE-2011-0611

Microsoft ATL/MFC ActiveX Type Confusion Vulnerability
CVE-2009-2494

Microsoft Office Excel Conditional Expression Ptg Type Confusion Vulnerability
CVE-2011-1989

The list goes on… and on… and on…
Reverse engineering C++ binaries is hard
As it turns out, these are all the same problem…

- Recently, many software-level vulnerabilities caused by C++ type confusion
- Compiled C++ code can be very difficult to analyze
  - IDS/IPS vendor wanted to provide signature coverage
  - Software consumer concerned with application security
  - Third-party interoperation
- Software developers regularly incorrectly use the `static_cast` operator
  - No compiler warning from most modern compilers
  - C++ standard only requires “cv-check”
Root of the problem

- This code compiles without warning with Visual Studio and g++ (< 4.6)
- Running this code causes a call to arbitrary memory
Same problem

• In the previous slide, the problem should be obvious to a developer
• Consider this code. _tmain() and internalFunction() may be “miles apart”
  • Separate libraries
    • Not caught by g++ 4.6
• Very common code construct in MS COM

```c
int internalFunction(void *pv)
{
    static_cast<class1*>(pv)->addRef();
    static_cast<class1*>(pv)->print();
    static_cast<class1*>(pv)->debug();
    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    class1 *C1 = new class1;
    class2 *C2 = new class2;
    internalFunction((void *)C1);
    internalFunction((void *)C2);
    return 0;
}
```
Structure of a C++ object after compilation

```cpp
00402138 off 402138 dd offset sub 4010D0
0040213C dd offset sub 4010A0
00402140 dd offset sub 4010D0
00402144 dd offset sub 4010A0
00402148 align 8
0040214C 48h ; H
00402150 db 48h ; H
00402154 db 0
00402158 db 0
0040215A db 0
0040215B db 0
0040215C db 0
```

Class2
- ptr to VTable
- prop1
- prop2
- ...

VTable
- void (Class1::*AddRef)()
- void (Class1::*print)()
- void (Class2::*voidFunc1)()
- void (Class2::*debug)()

```cpp
void Class1::AddRef()
{
    prop1++;
    return;
}

void Class1::print()
{
    cout << "I'm in Class1" << endl;
    return;
}

void Class2::*voidFunc1()
{
    return;
}

void Class2::*debug()
{
    cout << "In debug" << endl;
    return;
}
```
RECALL

• Reconstruct C++ objects from binary code

• Perform reaching definition analysis on object definitions to determine which object is being referenced at a given use point (make reverse engineering easier)

• Perform a “congruence check” to determine the safety of the use of a given object (detects vtable escape vulnerabilities)
High-level architecture of RECALL

IDA Pro → LLVM bitcode → ClassTracker

Assembly → LLVM IR → opt

x86 Machine Code → llvm-bcwriter

Resolved Methods → Type Mismatches
x86 to SSA

• First, we translate x86 machine code into an SSA-based IR
• We chose an SSA-based IR to make translation simpler
  • x86 assembly is mostly triple-based
  • Use-def chains are implicit (core requirement for reaching definitions)
  • Problems with going to higher-level IR
• Chose the LLVM IR due to the robustness of the LLVM analysis framework
• LLVM is attractive from a licensing perspective
Object reaching definition analysis

\[ REACH_{IN}[S] = \bigcup_{p \in \text{pred}(S)} REACH_{OUT}[p] \]

\[ REACH_{OUT}[S] = GEN[S] \cup (REACH_{IN}[S] - KILL[S]) \]

Where:

- **GEN** is the set of objects that are instantiated in a given basic block.
- **KILL** is the set of objects that are deleted in a given basic block.

For interprocedural analysis, \( REACH_{IN} \) at the entry of a function \( F \) is equal to \( REACH[c] \) at the call to \( F \) from a call site \( c \).
Indentifying object instantiation

• Stack-allocated
  
  Implement object structure heuristics
  
  • Inline constructor
  
  • Explicit constructor

• Heap-allocated – new() operator

  Call to YAPAXI(uint size)

  • Inline constructor
  
  • Explicit constructor
Tracking object types

- For each object, create a structure mapping the structure of the object
- Tag each object type with the virtual address of the constructor
Congruence Check

Class X
ptr to VTable
prop1
prop2
...

VTable
void (Class1::*AddRef)()
void (Class1::*print)()
void (Class2::*voidFunc1)()
void (Class2::*debug)()

void Class1::AddRef()
{
    prop1++;
    return;
}

void Class1::print()
{
    cout << "I'm in Class1"
        << endl;
    return;
}

void Class2::voidFunc1()
{
    return;
}

void Class2::debug()
{
    cout << "In debug"
         << endl;
    return;
}

Do these align?

Class Y
ptr to VTable
prop1
prop2
...

VTable
void (Class1::*AddRef)()
void (Class1::*print)()
void (Class2::*voidFunc1)()
void (Class2::*debug)()

void Class1::AddRef()
{
    prop1++;
    return;
}

void Class1::print()
{
    cout << "I'm in Class1"
        << endl;
    return;
}

void Class2::voidFunc1()
{
    return;
}

void Class2::debug()
{
    cout << "In debug"
         << endl;
    return;
}

Do these align?
Caveats

• Not designed for the analysis of malware or obfuscated code
• Does not require RTTI or debug symbols
• Focus is on code compiled with Visual Studio, but techniques can be generalized to other compilers
• If an object is allocated and the class pointer is stored in a collection, when the pointer is retrieved, we cannot track the type (future work)
Results

• Able to reconstruct and analyze objects from sample code that models:
  
  [stack-allocated, heap-allocated] x [inlined ctor, explicit ctor]

• Able to identify vulnerabilities in microbenchmarks designed to simulate real vulnerabilities:
  • Simulated CVE-2011-0611 (Adobe Reader)
  • Simulated CVE-2010-0258 (Microsoft Excel)
Why microbenchmarks?

- Analysis is performed interprocedurally
- Procedures can be analyzed independent of their location in the binary
- “Moving” procedures does not impact the correctness of the analysis

```c
Function_A:
...
%3 = new()
...
%6 = new()
...
delete(%6)
...
call Function_B(%3)
```

```c
Function_B(void* a):
...
%2 = new()
...
%5 = new()
...
delete(%2)
...
Call [a+0x4]
```
Select Related Work


Conclusion

• In our paper, we make the following contributions:
  - Resolve vtable dispatch calls in compiled binaries
  - Programmatically identify vtable escape vulnerabilities introduced by C++ developers
  - Construct a general C++ decompilation framework for use in other analyses
Questions?

ddewey@gatech.edu
giffin@gatech.edu