Improving C++ Virtual Calls Protection in the Code-Pointer Separation Mechanism

Motivation

Code-Pointer Separation [1] (CPS) is a new technique that provides protection against control-flow hijack attacks for programs written in C/C++, without imposing undue performance overhead. CPS separates code pointers from the rest of data in program memory and ensures that code pointers cannot be directly modified by corrupt non-code-pointer memory accesses, which prevents an attacker from forging code-pointers.

In presence of indirection, an attacker might be able to cause a CPS-protected program to use different code pointer than what the programmer intended (as long as that pointer was previously taken and stored in memory). Normally, this flexibility is not enough to perform a control-flow hijack attack, but it might be enough to trick a program to perform certain undesirable actions, especially when the number of functions that are called indirectly is large. CPI policy prevents this at a cost of imposing higher performance overhead then CPS.

One common example of programs with large number of indirectly callable functions are programs written in C++ that use virtual functions. Fortunately, C++ type-based rules for virtual function calls restrict the set of possible targets of each such call to a limited set, as determined by class inheritance. Enforcing these rules at runtime, as proposed by VTV [2], significantly reduces the flexibility of an attacker to change the control flow of a program. Combined with CPS, such protection could provide extra security guarantees for C++ programs that use virtual functions extensively. Moreover, we believe that such virtual function call protection can be enforced without imposing much overhead (in particular, with lower overhead than the original VTV [2]).

Project Goal

Implement low-overhead virtual call protection mechanism on top of CPS, as described above. The mechanism should ensure that, despite any memory corruptions during program execution, at each virtual function callsite that calls a method f() of class C (according to the static type of the callsite), the program may only call f() itself or any method that overwrites f() in some subclass of C.

The key idea behind fast implementation of virtual call protection mechanism is to place virtual tables in memory in a such a way that would enable performing the check described above without any extra memory accesses (in fact, two comparisons and a bitmask should be sufficient). This idea is to be discussed in person at the start of the project.

Evaluation criteria

The following provides an overall evaluation criteria for the project. Passing criteria is required to get 4, excellence criteria is required to get 6. Note that each milestone outlined below will be graded separately and the final grade will be averaged.

Passing criteria: Test the implemented protection mechanism on SPEC2006 benchmark and demonstrate that, for all benchmarks written in C++ that do not mix multiple inheritance with virtual functions:

1. The benchmarks successfully compile with the protection mechanism when the link-time optimizations are enabled and run without errors on SPEC workloads.
2. The mechanism prevents at least 5 manually injected vulnerabilities.
3. The overhead of the protection on the “ref” workload is less than or equals to the reported overhead of VTV [2].

Excellence criteria:

4. The overhead of the protection mechanism is within 1-2% (or explain why this is not possible to achieve).
5. The mechanism supports separate compilation (i.e., SPEC can be compiled without link-time optimizations).
6. The mechanism fully supports multiple inheritance.

Bonus:

7. The mechanism works on Phoronix benchmark that are used in the CPI paper [1].
8. The mechanism prevents at least one real exploits that overwrite virtual table pointers.

Schedule

Milestone 1: Learn how clang represents C++ classes and virtual tables [due on 17.11.2014]

Deliverable: a tool that can print class hierarchy and all virtual function callsites along with static type information for C++ programs.

1. Discuss the architecture of the protection mechanism to be implemented
2. View clang AST generated for simple C++ programs with classes and virtual functions
3. Print C++ class hierarchy based on clang AST
4. Print all virtual function callsites in the program, along with the information about static type of the object whose method is being called
5. Find out when and how virtual tables are generated by clang, print all virtual tables during compilation

Milestone 2: Pass information about virtual tables to LLVM [due on 27.11.2014]

Deliverable: a set of modifications to Clang that make it emit all information required to implement the virtual calls protection mechanism in the LLVM IR.

1. Decide which information about class hierarchy, virtual tables and virtual calls is necessary in order to compute and enforce virtual tables layout and insert checks as required for the protection mechanism.
2. Find out which of these information is already present in LLVM IR, and which needs to be added. Discuss this findings in person.
3. Modify clang to preserve necessary information when it emits LLVM IR.
4. Make sure that the modified clang successfully runs on all SPEC2006 benchmarks and manually verify its correctness on a sample from the benchmarks.

Milestone 3: Enforcing virtual tables layout [due on 11.12.2014]

Deliverable: an LLVM pass that enforces virtual tables layout as required for the virtual calls protection mechanism.

1. Implement an LLVM pass that would enforce the layout of the virtual tables that is required to implement fast checks for the protection mechanism. For now, assume that the pass runs at link-time optimizations phase and has complete information about all classes and virtual tables in the entire program.
2. Schedule the pass to run during link-time optimizations (e.g., similar to the CPI pass or the SafeStack pass).
3. Make sure that clang/llvm with the pass being implemented can successfully compile all SPEC2006 benchmarks and the resulting benchmarks run correctly. Manually verify the correctness of the pass on a sample from the benchmarks.

Milestone 4: Inserting runtime checks [due on 25.12.2014]

Deliverable: an LLVM pass that inserts runtime checks after virtual table pointer loads that verify the correctness of the loaded pointer according to the static type-based rules.

1. Implement the check insertion LLVM pass.
2. Schedule the pass to run during compilation or link-time optimization.
3. Make sure that clang/llvm with the pass being implemented can successfully compile all SPEC2006 benchmarks and the resulting benchmarks run correctly. Manually verify the correctness of the pass on a sample from the benchmarks.

Milestone 5: Test correctness and measure the performance overhead [due on 08.01.2015]

Deliverable: at least 5 exploits for manually injected vulnerabilities that the implemented protection mechanism successfully prevents; a graph that shows the performance overhead of the protection mechanism on all SPEC2006 benchmarks.

1. Manually inject 5 vulnerabilities in any of the SPEC2006 benchmarks and write corresponding exploits. Demonstrate that the implemented protection mechanism prevents the exploits form succeeding.
2. Measure the performance overhead of the protection mechanism on SPEC2006 and build the corresponding graph. If the overhead is higher than specified in the acceptance criteria, identify the sources of the overhead and optimize the implementation accordingly.

Milestone 6: Everything else [due on 22.01.2015]

Deliverable: everything that is required to meet the excellence criteria and (potentially) bonus criteria.

1. Further reduce the performance overhead of the instrumentation.
2. Add support for separate compilation (without link-time optimizations).
3. Add support for multiple inheritance.

References

[1] Kuznetsov, Volodymyr, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-pointer integrity." In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pp. 147-163. USENIX Association, 2014.

[2] Tice, Caroline, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. "Enforcing forward-edge control-flow integrity in GCC & LLVM." In *USENIX Security Symposium*. 2014.