Source Code-based Protection Against Bad Object Casts in C++ Applications

M.Sc. thesis problem statement
As the, 1) threat potential of advanced Code-Reuse Attacks (CRAs) is rising [4], 2) memory corruptions can not be completely ruled out from programs developed in unsafe languages (i.e., C, C++, etc.), and 3) the Clang+LLVM compiler can not detect any object bad casting during compile or runtime, we want to develop a tool that can mitigate such memory corruption vulnerabilities which are commonly the first step towards CRAs.

Based on source code recompilation techniques we want to harden the application binary and add the lowest possible performance penalty such that bad object casting in C++ applications are ruled out.

Previous work (e.g., Haller et al. [6] and Lee et al. [5]) focused on preventing bad object casts using the runtime checks which added high runtime overhead. In this work we want to protect against bad object casting which can lead to memory corruptions by using the class and virtual table hierarchy. Range checks should be added (with lowest possible runtime penalty) before each object cast such that only valid casts should be possible during program runtime.

The new bad object casting technique will be based on prior work in which we precisely secured indirect forward edges. Based on this work we want to implement a novel technique which mitigates bad object casts (i.e., similar to what UbSan [8], Caver [5] and TypeSan [6] do but more precisely and/or with lower runtime penalty).

The tool will be based on one or more LLVM instrumentation passes and will be tested with real CRAs for Linux/Windows OSs. Additionally, we will test the tool (by recompiling) with a series of server applications, web browsers and SPEC CPU 2006 benchmark w.r.t. performance.

Requirements
Strong C/C++ programming skills, LLVM pass knowledge is beneficial

Contact
Paul Muntean, M.Sc. E-Mail: paul@sec.in.tum.de, Tel.: (089) 289-18566, Tender date: September 29, 2016, Beginning: now
**Work Plan**

1. Develop knowledge of state-of-the-art source code hardening tools against advanced code-reuse attacks:
   
   (a) Read references [1, 7, 2, 3] and find related work on this topic.
   
   (b) Identify source code hardening tools suitable for real life software (e.g., web browsers, servers, etc.).
   
   (c) Write a state-of-the-art survey (max 5 A4 pages), which presents and compares the investigated techniques and tools.

2. Perform a security analysis of the CRAs mentioned in the reference:
   
   (a) Identify the binary assets/parts (e.g., indirect calls, vTables, class hierarchies (normal and virtual)) that need to be protected.
   
   (b) Identify, evaluate and perform attacks to extract the assets from the compiled binary.
   
   (c) Identify countermeasures based on binary hardening. What do the mentioned attacks violate? (hint: CFI). (e.g., number of function parameters, types, void function called were non void was expected, etc.)

3. Implement the program binary hardening technique which you identified and you think would make sense in order to prevent against bad object casts. Basically any down-cast or up-cast (e.g., dynamic, static) is a potential candidate for your tool.
   
   (a) Choose technique(s) described in literature and/or propose a new technique; argument your choice (e.g. security versus cost trade-off) in written form.
   
   (b) Implement the chosen technique(s) based on the LLVM framework \(^1\) and document design decisions.
   
   (c) Note: we provide a LLVM pass on which the tool can be build upon.

4. Evaluation of own implementation and possibly existing tools (case-study):
   
   (a) Measure effectiveness of your hardening tool against the same attacks identified in step 2.
   
   (b) Measure performance, effectiveness and size impact of the hardening on the SPEC 2006 benchmark, a series of server applications (e.g., Nginx, vfptd, lighttpd, etc.) and web browsers binaries (e.g., Chrome, Firefox, IE), by recompiling these open source apps.
   
   (c) Measure the performance of the hardened binary and of the transformation/tool itself.
   
   (d) Analyze and discuss security versus performance trade-offs.

5. The final thesis document must contain:
   
   (a) Description of the problem and motivation for the chosen approach
   
   (b) State-of-the-art survey, including analysis of security and performance
   
   (c) Security analysis of the previous mentioned server applications and web browsers
   
   (d) Rationale for choosing certain technique(s) for implementation
   
   (e) Implementation description
   
   (f) Performance evaluation of implementation
   
   (g) Discussion on potential security and performance trade-offs
   
   (h) Conclusions and future work.

**Deliverables**

1. Source code of the implementation (can be implemented as multiple LLVM passes) as well as instructions on how to run the tool.

2. Technical report with comprehensive documentation of the implementation, i.e., design decision, architecture description, API description and usage instructions.


**References**


[8] Clang undefined behavior sanitizer (UBSan).

\(^1\)http://llvm.org/