Compile Time Randomization

Colt VanWinkle, sa23885

Andy Davis, an24021

Abstract

This project proposes a compile-time system for exploit mitigation that is easy to use and transparent for developers. The provided mitigations are an improvement on current exploit mitigation techniques, such as ASLR and are compatible and intended to be used in conjunction with current OS level protections.

Problem

Current exploit mitigation techniques attempt to introduce uncertainty in the exploit process. Address space layout randomization (ASLR) is a prime example of this. By moving the body of code it is possible make code reuse attacks unfeasible. However, with current implementations, if one address is disclosed to an adversary, the adversary then knows the location of all code for a module, allowing its reuse in an exploit using ROP.

Additionally, the protections currently offered are often driven by the operating system designers; whom have a fiscal disincentive to 'break' any customer's software, even for security purposes. As such, the mitigations used are 'transparent' on part of the developers for a platform, requiring no retooling on their part, but often do not provide the best security possible.

Our solution is to address both the 'friendliness' of current techniques by requiring little effort on the part of developers, and to improve the strength of current techniques, specifically ASLR in defending against attacks that rely on ROP.

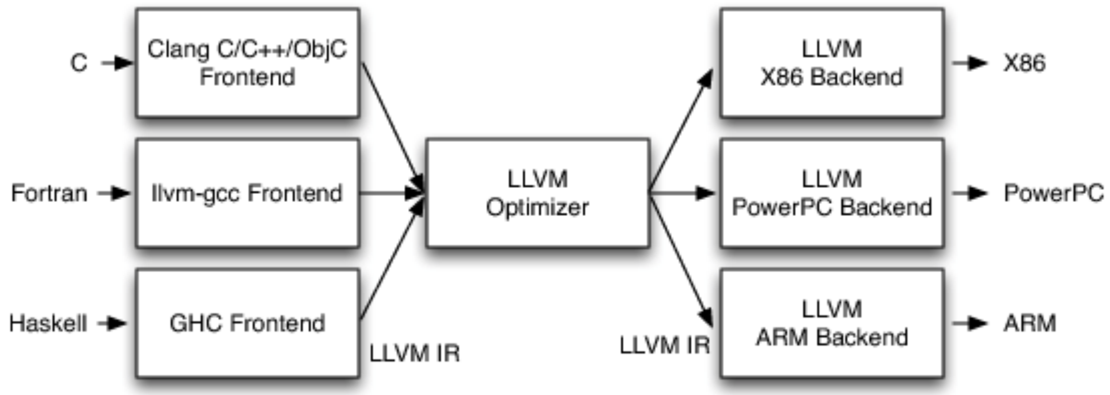
Solution

The solution provided expands on the concept of address space randomization. In that, the location of objects by an adversary becomes nondeterministic for a specific target user or server; even with an address disclosure for a module. The improvements provide a higher resolution of randomization than current ASLR implementations that depend on the operating system loader to move and 'fix-up' a module at run-time. Additionally, randomization is extended to the layout of data used by an application, specifically the stack and data sections. These techniques have been discussed in research [1], but these techniques were used in addition to several OS level changes.

Implementation

Our approach is to introduce a 'transparent' step during application compilation that provides the described protection features. To provide this higher resolution mitigation technique, it will be required that a per-install compilation be made. It is the authors feeling that this is an acceptable choice for systems running critical services. To improve on the 'user' experience, this system could be added to automated software deployment systems.

These goals are accomplished through the use of the LLVM compiler framework [2]. This framework allows for us to implement our protections for any language LLVM supports on any platform LLVM supports. The framework is divided in to what amounts to three phases, as illustrated below:



The frontend will consume a higher level language and produce a specialized intermediate representation (IR), denoted as LLVM IR. This IR can then be acted on by optimization routines. Finally, the IR is passed to a backend where it is transformed in to native instruction code for a target platform. Of note, the IR is inherently platform agnostic. Our transformations happen in the middle, as an 'optimization'.

Specific to the optimization methods in LLVM, there exists the concept of a 'Pass'. The passes represent the level of introspection depth to the code under analysis. A pass exists for each hierarchical object of an application. Specifically, a 'Module' pass, that describes the application as a whole; a 'Function' pass that is a representation of each function in a 'Module; and finally a 'Basic Block' pass that maps to each basic block of the control flow graph for all functions in a module.

The exploit mitigations described operate on the application in question via these optimization passes, performing actions on each modules, function, and basic black to introduce the desired effect.

We implemented fived different transforms: data section randomization, dead function insertion, block splitting, stack randomization, and NOP block insertion. The data section randomization pass randomizes the calling convention, stack alignment, inline tagging, and function alignment. The performance impact of this passes is hard to determine, but is most likely due to interfering with the processors optimal alignments. Dead function insertion creates a dummy function in each module containing a random number of instructions. This function is never called so the only performance impact should be the result of shifting of other function, which could interfere with caching. The block splitting pass iterates over the basic blocks in a function and randomly splits the block into two separate blocks. This pass has very strong randomization properties, but introduces significant performance overhead due to the extra branch instructions and increased code size. The next two passes are similar, but have slightly different properties. The stack randomization pass randomly adds extra stack allocations to the beginning of each function. This has two benefits. It adds random padding on the stack, which should make stack, based exploitation harder and the extra allocation instructions randomize the function. The perform impact is primarily the execution of the extra allocation functions and the impact of the larger stack. The NOP block insertion pass creates a random basic block and inserts it at the beginning of the function. The block is inserted at the beginning so as to not interfere with any of LLVM's block ordering optimizations. The performance impact of this pass is similar to the random stack allocation, but does not change the size of the stack.

Evaluation

To evaluate our passes we tested them on the SQLite project[3]. The SQLite project is a popular open source, embeddable SQL database system. We chose SQLite due to the projects extensive tests. The project contains 1084 times as much testing code as actual database code[4]. We tested our system on three characteristics: correctness, speed, and percentage of ROP gadgets moved.

For correctness, we ran SQLite's included test cases and ensured that they still passed. For speed, we timed the execution of the tests and compared them to a build with no passes applied. For the percentage of ROP gadgets moved, we used an open source ROP gadget finder[5] and compared the list of gadgets found to unmodified and modified binaries.

Results

Technique	Gadgets Moved (%)	Time Increase
Data Section	**75.098909**	**1.014**
Data Function	**98.092417**	**1.1388**
Block Splitting	**99.083074**	**1.45**
Stack Randomization	**99.117261**	**1.2088**
NOP Block	**98.989593**	**1.201**

References

[1] http://www.cs.vu.nl/~giuffrida/papers/usenixsec-2012.pdf

[2] http://llvm.org/

[3] http://www.sqlite.org

[4] http://www.sqlite.org/testing.html

[5] https://github.com/0vercl0k/rp