Poster: Data Confidentiality and Integrity

Scott A. Carr
carr27@purdue.edu
Purdue University

Mathias Payer
mathias.payer@nebelwelt.net
Purdue University

***Abstract*—The lack of memory safety in C/C++ results in an unending stream of security vulnerabilities. Unfortunately, only weak protection mechanisms are widely deployed in practice, because strong mechanisms cause severe overhead. Further, researchers have focused on preventing control-flow hijack attacks, leaving the door wide open for non-control data attacks – which only leak or alter non-control data. The recent HeartBleed Bug found in OpenSSL emphasizes the gravity of non-control data attacks. In this work, we propose a data protection mechanism we call Data Confidentiality and Integrity (DCI). As a compiler based approach, DCI provides a set of light-weight annotations to indicate which variables are sensitive and should be protected. The DCI framework rewrites the program to prohibit unannotated code from accessing protected variables. We evaluate our prototype implementation using a case study on PolarSSL. Our DCI prototype shows lower overhead than SoftBound+CETS, a complete protection mechanism, on the PolarSSL test suite.**

I. Introduction

In systems software, which is predominantly written in unsafe languages such as C/C++, the main attack vector is through memory safety errors. An attacker designs malicious inputs to trigger memory errors during program execution, allowing the attacker to read or write the program's memory. Examples of memory errors include buffer overflows, use-after-free, and use of uninitialized memory.

Research into mitigating these attacks has focused on preventing control-flow hijack attacks. In a control-flow hijack attack, the attacker exploits a memory error to divert the control-flow of the executing program from the programmer's intended control-flow. The attacker might make the program execute injected shell code, return to a specific libc function with a malicious argument [12], or setup gadgets for return orient programming [9]. ASLR [10] and Stack Cookies are widely used, but can be defeated. More robust protection mechanisms such as Control-Flow Integrity [2], SoftBound+CETS [6], [7], or modifications to the C language (i.e. CCured [8], and Cyclone [3]) have been proposed, but none have seen wide adoption due to prohibitive performance overhead.

Data Confidentiality and Integrity (DCI) takes a different approach and offers *strong* guarantees for a small *subset* of data. Programmers mark sensitive variables through type annotations and a generalization of Code-Pointer Integrity [4] protects both the integrity and confidentiality of all sensitive data by ensuring that unprotected data cannot access protected data and protected data is integrity protected at all times. We enforce this policy at runtime by creating a separate memory region for protected data and enforcing bounds for all object accesses in that area. At compile time, our compiler rewrites unannotated code to forbid it from accessing the protected region. Allocations of protected data are rewritten to allocate to the protected region as well as create bounds information for protected memory objects.

II. Threat Model

Our threat model gives the attacker capabilities to read and write any location in the program's data memory. We assume the code segment is protected by a mechanism such as write-or-execute. This model represents the case where the attacker exploits memory errors in the program. Since control-flow hijack attacks are well-researched, they are out of scope for this work and we assume that DCI is combined with another protection mechanism that provides control-flow protection.

III. Data Confidentiality and Integrity

Our work is an extension of Code Pointer Integrity (CPI) [4]. Retrofitting complete memory safety on existing low level languages results in prohibitively high overhead, but protecting an important subset allows for a configurable security to overhead trade off. For CPI, only code pointers (return addresses, function pointers, etc.) are protected. In our work we extend this protection to a programmer selected subset of all data types. We hypothesize that there is only a small amount of data in a program that is truly sensitive. Examples of sensitive data include encryption keys, password lists, and authentication tokens. However, there is a large amount of data that is not sensitive. Our work provides a framework where the programmer can control the overhead to protection trade off. Our DCI framework has three main components:

1) A set of light-weight annotations the programmer uses to specify the sensitive data.
2) A compiler plug-in that reads the protection specification and rewrites the program to ensure protection of sensitive data.
3) A runtime for enforcing the policy and maintaining the necessary metadata.

IV. Implementation

At a high level, our prototype implementation is inspired by Software Fault Isolation. The idea is that the unannotated (unprotected) variables and code should be isolated from the protected data. This is accomplished by moving all the protected data into a separate memory region and enforcing its bounds. Additionally, protected variables are bounds checked, preventing overflows within the protected region.

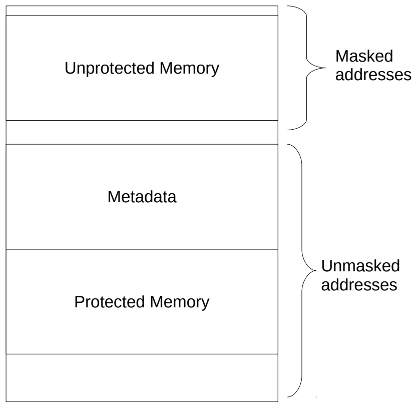

Fig. 1: Memory layout of the DCI mechanism.

DCI's annotations apply to types, not variables. We made this design decision to make annotating large code bases as painless as possible. Annotating types allows one annotation to apply to many variables throughout the program whereas annotating individual variables would result in a lot more typing. Annotation burden is a major issue for any tool that requires programmer annotations [11].

Our prototype DCI implementation is built on top of the LLVM compiler infrastructure [5]. The analysis and rewriting step is implemented as a module pass.

The analysis step identifies annotated variables and verifies the isolation of protected and unprotected variables. Identifying protected variables is accomplished by searching the LLVM IR for the annotation metadata. After identifying protected variables, DCI then performs conservative data-flow analysis to ensure there is no data flow between protected and unprotected variables. An interesting design point is what DCI should do if it discovers illegal data-flow (either from protected to unprotected or vice versa). A simple approach aborts the compilation and prints an error message. However, we also implemented the option to automatically promote the unprotected variable to protected. This approach increases the runtime overhead (more protected variables result in more runtime checks and more metadata) but further reduces annotation burden. With automatic promotion the programmer can just make a few seed annotations to a large program, then DCI will automatically annotate any variable that interacts with them. From a security perspective it is always safe to protect a variable since protected variables are always bounds checked. Our analysis is conservative and errs on the side of protecting too many variables (possibly leading to higher overhead). In future work, we will investigate other approaches for identifying protected variables and providing feedback to the programmer on the size of the set of protected variables.

The rewriting step ensures that only annotated variables can access the protected region. Loads and stores to protected variables are rewritten to use our DCI intrinsic functions and allocations of protected memory objects are written to use our intrinsic allocation functions. Values from unprotected pointers are masked before they are dereferenced to prevent them from accessing protected data. Figure 1 shows the memory layout.

The runtime component is integrated into LLVM's libclang. It maintains the data structure necessary for protected variable bounds checks. The current implementation uses a lookup table to hold bounds information for a pointer's address.

V. Case Study

To investigate the practicality of DCI we applied it to PolarSSL [1], a SSL/TLS library implemented with 30,000 lines of C code. We protect the *very* frequently used type `ssl_context`. Using our instrumented PolarSSL we ran the provided client and server to verify correct functionality.

Besides the client and server examples, PolarSSL has a large test suite of 6,212 tests. Our protected PolarSSL library passes all tests. PolarSSL includes a benchmark with 84 separate measurements. For benchmarking, we annotated the `arc4_context` type to make it a protected type. We measured an average overhead of 7.28x on the benchmarks. For comparison, we also ran the benchmark with SoftBound+CETS and measured an overhead of 11.44x. As DCI protects less data the overall overhead is reduced compared to SoftBound+CETS. These results are very preliminary, we plan to thoroughly evaluate our prototype in future work.

VI. Conclusion

Data Confidentiality addresses the emerging problem of non-control data attacks. Since these attacks do not alter the intended control-flow they will slip by existing protection mechanisms. Even though our implementation is still under development our preliminary results show the approach is practical for large realistic programs and has lower overhead than mechanisms that enforce complete memory safety.

VII. Acknowledgements

This research is supported by the DHS STEM Career Development Program Fellowship, NSF, and Purdue Research Foundation.

References

[1] PolarSSL. https://polarssl.org.
[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS'05*, 2005.
[3] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Usenix ATC'02*, 2002.
[4] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI'14*, 2014.
[5] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, 2004.
[6] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI'09*, 2009.
[7] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *ISMM'10*, 2010.
[8] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe Retrofitting of Legacy Software. *TOPLAS'05*, 2005.
[9] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58):http://phrack.com/issues.html?issue=67&id=8, Nov. 2007.
[10] PaX-Team. PaX ASLR (Address Space Layout Randomization). http://pax.grsecurity.net/docs/aslr.txt, 2003.
[11] T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *ICSE'14*, 2014.
[12] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). 2007.