Errata Slip

Proceedings of the 26th USENIX Security Symposium

In the paper "Efficient Protection of Path-Sensitive Control Security" by Ren Ding and Chenxiong Qian, *Georgia Tech;* Chengyu Song, *UC Riverside;* Bill Harris, Taesoo Kim, and Wenke Lee, *Georgia Tech* (Wednesday session, "Systems Security I," pp. 131–148 of the Proceedings) the authors wish to add the following:

Acknowledgements

We thank our anonymous reviewers for their helpful feedback. This research was supported by the NSF under award DGE-1500084, CNS-1563848, CRI-1629851, CNS-1017265, CNS-0831300, and CNS-1149051, ONR under grant N000140911042 and N000141512162, DHS under contract No. N66001-12-C-0133, United States Air Force under contract No. FA8650-10-C-7025, DARPA under contract No. DARPA FA8650-15-C-7556, and DARPA HR0011-16-C-0059, and ETRI under grant MSIP/IITP[B0101-15-0644].

In the paper "Venerable Variadic Vulnerabilities Vanquished" by Priyam Biswas, *Purdue University;* Alessandro Di Federico, *Politecnico di Milano;* Scott A. Carr, *Purdue University;* Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, and Michael Franz, *University of California, Irvine;* Mathias Payer, *Purdue University* (Wednesday session, "Bug Finding II," pp. 183–198 of the Proceedings) the authors wish to add the following:

Programming languages like C and C++ support variadic functions (i.e., functions with variable number of arguments). For such functions the semantic depends on the implicit agreement between caller and callee. In "Venerable Variadic Vulnerabilities Vanquished" [1], we propose HexVASAN, a sanitizer that makes the contract between caller and callee explicit by type checking each function argument that is used in the callee.

For each variadic call, the caller generates a list of types that is then used in the callee to verify that the expected types match the actual types. In the paper, we describe the design of a detailed algorithm that statically hashes the types to a unique number.

In the initial prototype, we used this described algorithm. As we extended testing to additional software, we simplified the implementation of the algorithm for pointer types (to handle some then unsupported corner cases). This simplified implementation distinguishes between primitive types, unpacked aggregate types, and primitive pointers. The simplified implementation was unable to distinguish between different packed aggregate types (e.g., `struct {int, int}` and `struct {int, double}`) or sub-types of primitive types. After testing, we forgot to revert back to the full hashing algorithm.

To fully support all these corner cases, the implementation needs to be aware of the part of the ABI that covers packing/unpacking of aggregate types during calls. We have updated our prototype accordingly and extended the original implementation with support for packing/unpacking. Note that the inconsistency only affected the compile time of programs. We repeated the full evaluation to validate our updated prototype. We tested SPEC CPU2006 on a server system with an Intel Xeon E5-2660 CPU and 64 GiB of RAM, and Firefox on a desktop system with an Intel Core i7-4790 CPU and 16 GiB of RAM. Both systems run Ubuntu 14.0.5 LTS 64-bit. We tested Firefox 51.0.1 as well as 54.0.1, the latest version as of this writing. Figure 1 and Table 1 show the SPEC and Firefox results. As hashing is purely static, the performance remained roughly the same (as expected). Due to the increased precision, we discovered new type violations in Firefox.

We would like to thank Istvan Haller for the thorough external code review of our open-sourced prototype and for informing us and the PC chairs about the simplification in the original prototype. We acknowledge the inconsistency between the design section and the prototype used for the evaluation. We have updated the prototype so that the design section in the paper and open-source prototype implementation match fully.

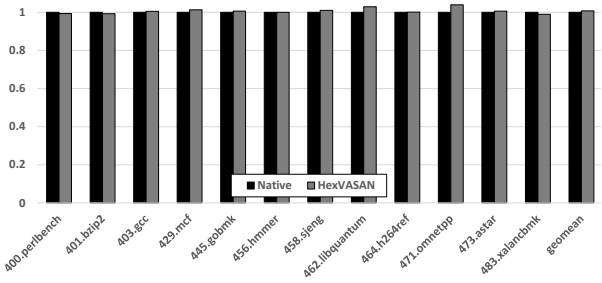

Figure 1: Runtime overhead of HexVASAN for the SPEC CINT2006 benchmarks, compared to baseline.

Benchmark		Native	HexVASAN
Octane	AVERAGE	33824.40	33717.40
	STDDEV	74.96	125.89
	OVERHEAD		0.32%
JetStream	AVERAGE	194.86	193.68
	STDDEV	1.30	0.58
	OVERHEAD		0.61%
Kraken	AVERAGE [ms]	885.52	887.12
	STDDEV [ms]	11.02	7.31
	OVERHEAD		0.18%

Table 1: Updated performance overhead on Firefox benchmarks. For Octane and JetStream higher is better, while for Kraken lower is better.

References

[1] BISWAS, P., FEDERICO, A. D., CARR, S. A., RAJASEKARAN, P., VOLCKAERT, S., NA, Y., FRANZ, M., AND PAYER, M. Venerable Variadic Vulnerabilities Vanquished. In *SEC: USENIX Security Symposium* (2017).