Weird Machines as Insecure Compilation

Jennifer Paykin, Eric Mertens, Mark Tullsen, Luke Maurer, Benoît Razet, and Scott Moore
{jpaykin,scott}@galois.com *Galois, Inc.*

Computer security is distinguished from other computer science disciplines by its adversarial nature—computer security studies how systems behave while subject to attack. A result of this adversarial focus is that *exploits* serve an important role in security research: an exploit witnesses the insecurity of a system by causing it to behave inappropriately.

However, the mere existence of an exploit fails to answer many important questions about the system under consideration: How severe is the vulnerability? How readily can the vulnerability be repurposed by hackers to attack other systems? Can the vulnerability be patched? How effective is a proposed mitigation? Without a systematic approach to understanding exploits, it is difficult to evaluate the importance of any particular vulnerability or to generalize lessons learned to improve security more broadly.

In the exploit community, many practitioners describe exploit development as an exercise in "programming a *weird machine*." A weird machine is the latent computational machine exposed by a vulnerable program that can be repurposed by an attacker to achieve their goals [1, 2]. A particularly evocative example of programming a weird machine is return-oriented programming, where attackers exploit a program by overwriting the stack with a sequence of return addresses that invoke fragments of the original binary to achieve a desired effect [3]. Despite the intuitive appeal of weird machines, it has proven challenging to provide a formal definition that can be consistently applied to a variety of systems and vulnerabilities [4, 5].

Dullien [5] defines weird behavior as the difference between two state machines—an *intended finite state machine* (IFSM) corresponding to the model that the programmer has in her head when writing the program and an implemented finite state machine that attempts to realize the IFSM. A weird state is a state in the implemented finite state machine that does not correspond to a state in the intended one, and the weird machine is the collection of computations reachable from a weird state. Using this formalism, Dullien is able to compare the exploitability of two implementations of a simple program.

This formalism requires that the relationship between the intended and actual state machines is formalized, but does not give clear guidelines for what this relationship must look like. Furthermore, since both the intended and implemented state machines are program-specific, it is difficult to draw conclusions about the generalizability of exploits or mitigations beyond the system at hand.

Arguing for the development of new formalisms of weird machines, Bratus and Shubina [2] identify abstraction-breaking as their core phenomenon. Inspired by Abadi's [6] connection between programming language abstractions and secure compilation, we propose formalizing weird machines as counterexamples to secure compilation between a source program (written P) and its compilation (written $[\![\mathsf{P}]\!]$), witnessed by a target-level attacker context $\mathbf{A}$. Following Abate et al. [7], we define secure compilation via property-preservation, rather than full abstraction, explicitly modeling a program's behavior (written $\mathsf{Behav}\ (\mathsf{P})$).

An exploit is an adversarial context that causes a compiled program to behave differently than it could in the source-language semantics. Intuitively, such a behavior must violate some abstraction of the source-level program. More formally:

Definition (Exploit)**.** *An* exploit *of a vulnerable source program* V *is an attacker context* $\mathbf{A}$ *from an attack class* $\mathcal{A}$ *if* $\mathsf{Behav}\ (\mathsf{C}[\mathsf{V}]) \neq \mathsf{Behav}\ (\mathbf{A}[[\![\mathsf{V}]\!]])$ *for every non-oblivious*[1] C*.*

$$\mathit{Exploit}^{\mathcal{A}}(\mathsf{V}) \triangleq \left\{ \mathbf{A} \in \mathcal{A} \;\middle|\; \begin{array}{l} \forall\ \mathsf{C}\ .\ \neg \mathit{oblivious}(\mathsf{C}) \ \Rightarrow \\ \mathsf{Behav}\ (\mathsf{C}[\mathsf{V}]) \neq \mathsf{Behav}\ (\mathbf{A}[[\![\mathsf{V}]\!]]) \end{array} \right\}$$

Given a vulnerable program, the set of possible exploits determines what behaviors are available to the attacker:

Definition (Weird Machine)**.** *The* weird machine *of a vulnerable source program* V *for an attack class* $\mathcal{A}$ *is the collection of behaviors arising from exploits of* V*.*

$$\mathit{WM}^{\mathcal{A}}(\mathsf{V}) \triangleq \left\{ \mathsf{Behav}\ (\mathbf{A}[[\![\mathsf{V}]\!]]) \;\middle|\; \mathbf{A} \in \mathit{Exploit}^{\mathcal{A}}(\mathsf{V}) \right\}$$

Using these definitions, we model a number of weird machines and exploits including return-oriented programming [3], data-oriented programming [8], speculative execution vulnerabilities [9, 10], and timing side-channels. Furthermore, we show that our approach generalizes Dullien's state machine formalization. Finally, we show that exploits are exactly the contexts that violate robust properties of behaviors [7].

Formalizing weird machines in terms of secure compilation provides a clear framework for understanding exploit classes like these and, as evidenced by Abadi et al's previous work on address space layout randomization [11, 12], allows us to reason about the underlying causes of exploits and how they might be mitigated.

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-15-C-0124. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force and DARPA.

[1] A context is *oblivious* if its behavior does not depend on the program it is linked with: $\mathsf{oblivious}(C) \triangleq \forall \mathsf{V}, \mathsf{V}'\ .\ \mathsf{Behav}\ (\mathsf{C}[\mathsf{V}]) = \mathsf{Behav}\ \mathsf{C}[\mathsf{V}']$.

References

[1] S. Bratus, M. Locasto, M. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: from Buffer Overflows to Weird Machines and Theory of Computation," *USENIX*, Dec. 2011. [Online]. Available: http://langsec.org/papers/Bratus.pdf

[2] S. Bratus and A. Shubina, "Exploitation as code reuse: On the need of formalization," *Information Technology*, vol. 50, no. 2, 2017.

[3] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[4] J. Vanegue, "The weird machines in proof-carrying code," in *Proceedings of the 2014 IEEE Security and Privacy Workshops (LangSec)*, 2014.

[5] T. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2018.

[6] M. Abadi, *Protection in Programming-Language Translations*. Springer Berlin Heidelberg, 1999, pp. 19–34.

[7] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF)*, 2019.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[9] J. Wampler, I. Martiny, and E. Wustrow, "Exspectre: Hiding malware in speculative execution," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[10] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.

[11] M. Abadi and G. D. Plotkin, "On protection by layout randomization," *ACM Transactions on Information Systems Security*, vol. 15, no. 2, Jul. 2012.

[12] M. Abadi and J. Planul, "On layout randomization for arrays and functions," in *Principles of Security and Trust (POST)*, 2013.