Security and the Average Programmer[*] (Invited Contribution)

Daniel Giffin, Stefan Heule, Amit Levy, David Mazières,
John Mitchell, Alejandro Russo*, Amy Shen, Deian Stefan,
David Terei, and Edward Yang

Stanford and *Chalmers

Software security research spans a broad spectrum of approaches. At one end, experts attempt to build systems that are secure by construction. At the other end, people deploy faulty software and leave it to security practitioners to clean up the mess. But cleaning up the mess isn't working: experience shows that post-hoc fixes can't be deployed in time to prevent damage. Moreover, fixing faulty software is an arms race, and the security community shows no signs of winning it. Worse, the war is spreading to new fronts: even cars [5], televisions and refrigerators [2] are now vulnerable to network attack.

How can we make software secure from the start? For most software to be secure, the median programmer will have to produce secure code. Attempts to achieve this by building a culture of good security practices have met with limited success. For example, despite attempts to educate them, web programmers continue to misuse postMessage authentication [8]. Even Linux kernel developers have committed vulnerable code three times in a row for a single bug [9].

Rather than focus on the abstract notion of security culture, we argue it is more effective to change programmer behavior through APIs and programming languages. Designing APIs and programming languages with security in mind allows us to make common operations less error-prone, and, more importantly, to restrict the damage that leads from inevitable mistakes. This requires security mechanisms that, within the context of a single application, can protect programmers from themselves as well as from each other. What should such mechanisms look like?

To provide maximum benefit, any security mechanism must be objective: it should provide concrete, formally specifiable, and (in the event of a design error) falsifiable guarantees. Security mechanisms that evolve with systems tend not to have this property. For example, enforcement of the same-origin policy is split across multiple locations in Firefox—permission to load a resource is checked in a completely different place from iframe DOM access. Without a suitable security mechanism, the same-origin policy had to be expressed and enforced in a series

[*] This work was funded by the DARPA Clean-Slate Design of Resilient, Adaptive, Secure Hosts (CRASH) program, BAA-10-70, under under contract #N66001-10-2-4088 (*Bridging the Security Gap with Decentralized Information Flow Control*), as well as multiple gifts from Google.

of conditional statements. As software evolves with new features, extending such a regime in a consistent way becomes a subjective exercise.

An equally important property for a security mechanism is to capture real-world security concerns in a direct, declarative way. The issues people actually care about tend to be high-level questions—e.g., Who can see this photograph?—rather than low-level details—e.g., Does this image filter access the network? Ideally, the security mechanism can capture such policy concerns in a manner substantially divorced from the complex inner workings of an application.

One promising family of mechanisms is those based on decentralized information flow control, or DIFC [7]. DIFC allows one to specify policy in terms of who can read and write various data, and enforces these constraints throughout an application or system regardless of its structure or the sequence of operations performed. Specifying policy on data naturally captures high-level concerns in a direct and declarative way, fulfilling one of our criteria. (Indeed, the generality of DIFC is demonstrated by its ability to enforce policies uniformly across hardware [14, 1], operating systems [3, 12], programming languages [7], distributed systems [13, 6], and browsers [11, 10].) Moreover, DIFC guarantees can be formally specified (for example, as non-interference), fulfilling the other criterion.

Historically, two weak points of DIFC have been, first, the discrepancy between formal models and actual implementations (notably, where covert channels violate non-interference) and, second, limited adoption by non-experts. However, we have made progress on both fronts in recent years. This talk will report on our experience with Hails [4], a DIFC framework for building extensible web applications. Hails structures a web application as a collection of mutually distrustful "apps" and database policies. Hails has been used to build production web sites with minimal trusted code, making it one of the largest real-world examples of DIFC. Moreover, the system has been used by novices, giving us invaluable insight into the obstacles DIFC faces for adoption by average programmers.

References

1. de Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hriţcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. In: Proceedings of the 41st Symposium on Principles of Programming Languages, POPL (January 2014)
2. Bort, J.: For the first time, hackers have used a refrigerator to attack businesses. Business Insider (January 2014), http://www.businessinsider.com/hackers-use-a-refridgerator-to-attack-businesses-2014-1
3. Efstathopoulos, P., Krohn, M., van DeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R.: Labels and event processes in the Asbestos operating system, pp. 17–30
4. Giffin, D.B., Levy, A., Stefan, D., Terei, D., Mazières, D., Mitchell, J., Russo, A.: Hails: Protecting data privacy in untrusted web applications. In: 10th Symposium on Operating Systems Design and Implementation (October 2012)
5. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., Mccoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S.: Experimental security

analysis of a modern automobile. In: Proceedings of IEEE Symposium on Security and Privacy (2010)

6. Liu, J., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: A platform for secure distributed computation and storage. In: Proceedings of the 22nd Symposium on Operating Systems Principles, pp. 321–334 (October 2009)
7. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: 16th ACM Symposium on Operating Systems Principles, pp. 129–142 (1997)
8. Son, S., Shmatikov, V.: The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In: 20th Network and Distributed System Security Symposium (2013)
9. Wang, X., Chen, H., Jia, Z., Zeldovich, N., Frans Kaashoek, M.: Improving integer security for systems with KINT. In: 10th USENIX Symposium on Operating Systems Design and Implementation (October 2012)
10. Yang, E.Z., Stefan, D., Mitchell, J., Mazières, D., Marchenko, P., Karp, B.: Toward principled browser security. In: 14th Workshop on Hot Topics in Operating Systems (May 2013)
11. Yip, A., Narula, N., Krohn, M., Morris, R.: Privacy-preserving browser-side scripting with bflow. In: Proceedings of the 4th ACM European Conference on Computer Systems, pp. 233–246. ACM (2009)
12. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: 7th Symposium on Operating Systems Design and Implementation, Seattle, WA, pp. 263–278 (November 2006)
13. Zeldovich, N., Boyd-Wickizer, S., Mazières, D.: Securing distributed systems with information flow control. In: 6th Symposium on Networked Systems Design and Implementation, San Francisco, CA, pp. 293–308 (April 2008)
14. Zeldovich, N., Kannan, H., Dalton, M., Kozyrakis, C.: Hardware enforcement of application security policies using tagged memory. In: Eighth Symposium on Operating Systems Design and Implementation, San Diego, CA, pp. 225–240 (December 2008)