Software Security: A Formal Perspective
(Notes for a Talk)

Martín Abadi[1,2]

[1] Microsoft Research Silicon Valley
[2] University of California, Santa Cruz

Abstract. Weaknesses in software security have been numerous, sometimes startling, and often serious. Many of them stem from apparently small low-level errors (e.g., buffer overflows). Ideally, those errors should be avoided by design, or at least fixed after the fact. In practice, on the other hand, we may have to tolerate some vulnerabilities, with appropriate models, architectures, and tools.

This short paper is intended to accompany a talk at the 18th International Symposium on Formal Methods (FM 2012). The talk will discuss software security with an emphasis on low-level attacks and defenses and on their formal aspects. It will focus on systematic mitigations (specifically, techniques for layout randomization and control-flow integrity) that aim to be effective in the presence of buggy software and powerful attackers.

1 The Problem

Security depends not only on the properties of security models and designs but also on implementation details. Flaws, at any level, can result in vulnerabilities that attackers may be able to exploit. In the domain of software, those vulnerabilities often stem from small but catastrophic programming errors. For instance, buffer overflows remain frequent, and they can have serious consequences. An unchecked buffer overflow in a mundane parser can lead to the complete compromise of an operating system.

Although this short paper emphasizes low-level phenomena such as buffer overflows, similar considerations often apply for higher-level software. Indeed, attacks at all levels present common themes. For instance, attacks of various sorts often exploit errors in parsing and in sanitizing inputs in order to inject code into a target system.

Many errors may be fixed or avoided altogether by the use of suitable programming methods and tools. In particular, strong type systems may prevent the occurrence of many frequent flaws. The application of formal methods may further provide evidence of finer properties of software systems, or may establish essential properties of those system components that rely on low-level languages. For example, in this spirit, recent work [20] combines type safety and verification to obtain correctness guarantees for a research operating system.

However, to date, those methods and tools frequently fall short, in at least two respects:

D. Giannakopoulou and D. Méry (Eds.): FM 2012, LNCS 7436, pp. 1–5, 2012.

- Much code is still written in C, C++, and other low-level languages, often for performance or compatibility reasons. Full verification remains rare. Current tools for static and dynamic code analysis for those languages are remarkably effective, but still imperfect.
- Even code written in modern languages, and even verified code, should be treated with a healthy degree of caution. While type-safe programming languages like Java may well improve security, their implementations may not provide all the expected properties [1]. They have been the target of significant, successful attacks. For example, in 2012, malicious software called Flashback exploits a vulnerability in Java systems in order to install itself on Mac computers.

Moreover, implementation details, right or wrong, ultimately should be understood and judged in the context of security requirements. Programming methods and tools typically do not address how those requirements should be formulated. That is the role of security models (e.g., [12]). These models define precise security goals, with abstract concepts (such as principal and object) and properties (such as non-interference). Even though here we discuss them only briefly, these models are arguably crucial to software security. With the guidance of models, there is at least some hope that security measures are applied consistently and pervasively—they often are not.

We may ask, then, what is our fall-back position? Articulating and developing a tenable "Plan B" may be the best we can do, and quite useful.

2 Some Approaches

Despite their many flaws, software systems should guarantee at least some basic security properties, at least most of the time. Ideally, these guarantees should be obtained even in the absence of complete, precise security definitions and requirements, and even if many aspects of the systems are designed without security in mind. Although usually implicit, and perhaps still too optimistic, this point of view has motivated much recent work on architectures and tools.

In particular, the development of robust architectures may help confine the effects of dangerous errors. A system can sometimes be structured in such a way that a local compromise in one component does not immediately endanger the security of the entire system. This idea is not new. It appears, in particular, in classic work on mandatory access control, which aims to guarantee security even in the presence of Trojan horses [8].

Furthermore, the effects of flaws may be mitigated at run-time [6]. For instance, with the use of stack canaries, attacks that rely on buffer overflows can sometimes be detected and stopped before they take control of a target system [4]. These mitigations are often imperfect, and seldom enforce well-defined security properties. Nevertheless, these mitigations are based on useful insights on the goals and mechanisms of attacks. Their deployment has led attackers to shift their targets or to develop more elaborate techniques (e.g., [6,16,18]), in particular techniques that rely somewhat less bluntly on code injection.

Some advanced attacks include accessing data (e.g., communication buffers) or running code (e.g., library functions) that are present in the target system at predictable locations. One popular approach to thwarting such attacks consists in randomizing the placement of that data and code in memory (e.g., [5,15]); other types of randomization may also play a role (e.g., [7,14]). Recent research [3,9,17] shows that—at least in theory and in simple settings—randomization can yield precise guarantees comparable to those offered by the use of language-level abstractions.

More broadly, many attacks include unexpected accesses to data and unexpected control transfers. Run-time techniques for enforcing policies on data accesses and on control transfers can thwart such attacks (e.g., [2,10,11,13,19,21]). Recent research on these techniques has leveraged formal ideas and methods, defining models of machines, programs, run-time guards, and their objectives (which are typically safety properties), and then proving that those objectives are met.

Suppose, for example, that a piece of trusted code contains the computed-jump instruction `jmp ecx`, which transfers control to the address contained in register `ecx`. Let us assume that a programming mistake allows an attacker to corrupt or even to choose the contents of `ecx`. If the attacker knows the address `A` of another piece of code, and arranges that this address be placed in `ecx`, then it can cause that code to be executed. In the worst case, the code at address `A` could then give complete control to the attacker.

Since the attack has several hypotheses and steps, several possible countermeasures may be considered. These include:

1. We may identify and fix the mistake so that the attacker cannot tamper with `ecx`.
2. We may make the value `A` unguessable, by randomizing the layout of code in memory.
3. We may preface the instruction `jmp ecx` by a sequence of instructions that, dynamically, will check that `ecx` contains one of a set of expected values considered safe. (This set of expected values would be defined by a policy, perhaps inferred from source code.)

The first strategy is principled, but it may not always seem viable. Although the second and the third strategies also present non-trivial requirements, they are realistic and attractive in many low-level systems. Each strategy leads to different guarantees. For instance, with the third strategy, we obtain probabilistic properties at best.

Many variants and combinations of protection techniques are under consideration, often still with only preliminary analyses and prototype implementations. Further research may address the traditional goals of these techniques (e.g., proving isolation properties) and more delicate aspects of the subject (e.g., permitting controlled sharing). In this context, the application of formal ideas and methods will not lead to absolute proofs of security, but it can nevertheless be fruitful.

References

1. Abadi, M.: Protection in Programming-Language Translations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 868–883. Springer, Heidelberg (1998)
2. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-Flow Integrity: Principles, Implementations, and Applications. ACM Transactions on Information and System Security 13(1), 1–40 (2009)
3. Abadi, M., Plotkin, G.D.: On Protection by Layout Randomization. ACM Transactions on Information and System Security (to appear, 2012); The talk at FM 2012 will also cover an unpublished variant, joint work with Jérémy Planul
4. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the 7th Usenix Security Symposium, pp. 63–78 (1998)
5. Druschel, P., Peterson, L.L.: High-Performance Cross-Domain Data Transfer. Technical Report TR 92-11, Department of Computer Science, The University of Arizona (March 1992)
6. Erlingsson, Ú.: Low-Level Software Security: Attacks and Defenses. In: Aldini, A., Gorrieri, R. (eds.) FOSAD 2007. LNCS, vol. 4677, pp. 92–134. Springer, Heidelberg (2007)
7. Forrest, S., Somayaji, A., Ackley, D.H.: Building Diverse Computer Systems. In: 6th Workshop on Hot Topics in Operating Systems, pp. 67–72 (1997)
8. Gasser, M.: Building a Secure Computer System. Van Nostrand Reinhold, New York (1988)
9. Jagadeesan, R., Pitcher, C., Rathke, J., Riely, J.: Local Memory Via Layout Randomization. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, pp. 161–174 (2011)
10. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure Execution Via Program Shepherding. In: Proceedings of the 11th Usenix Security Symposium, pp. 191–206 (2002)
11. McCamant, S., Morrisett, G.: Evaluating SFI for a CISC Architecture. In: Proceedings of the 15th USENIX Security Symposium, pp. 15–15 (2006)
12. McLean, J.: Security Models. In: Marciniak, J. (ed.) Encyclopedia of Software Engineering. Wiley & Sons (1994)
13. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B., Gan, E.: RockSalt: Better, Faster, Stronger SFI for the x86. In: 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (to appear, 2012)
14. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the Gadgets: Hindering Return-Oriented Programming Using in-Place Code Randomization. In: IEEE Symposium on Security and Privacy, pp. 601–615 (2012)
15. PaX Project. The PaX project (2004), `http://pax.grsecurity.net/`
16. Pincus, J., Baker, B.: Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. IEEE Security and Privacy 2(4), 20–27 (2004)
17. Pucella, R., Schneider, F.B.: Independence from Obfuscation: A Semantic Framework for Diversity. In: 19th IEEE Computer Security Foundations Workshop, pp. 230–241 (2006)
18. Sotirov, A., Dowd, M.: Bypassing Browser Memory Protections: Setting Back Browser Security by 10 Years (2008), `http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf`

19. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient Software-Based Fault Isolation. ACM SIGOPS Operating Systems Review 27(5), 203–216 (1993)
20. Yang, J., Hawblitzel, C.: Safe to the Last Instruction: Automated Verification of a Type-Safe Operating System. Communications of the ACM 54(12), 123–131 (2011)
21. Yee, B., Sehr, D., Dardyk, G., Bradley Chen, J., Muth, R., Ormandy, T., Okasaka, S., Narula, N., Fullagar, N.: Native Client: a Sandbox for Portable, Untrusted x86 Native Code. Communications of the ACM 53(1), 91–99 (2010)