The Monoculture Risk Put into Context

Conventional wisdom holds that software monocultures are exceptionally vulnerable to malware outbreaks. The authors argue that this oversimplifies and misleads. An analysis based on attacker reactions suggests that deploying a monoculture in conjunction with automated diversity is indeed a very sensible defense.

Fred B. Schneider and Kenneth P. Birman
Cornell University

The term *monoculture* originates in the biological sciences, where it refers to a population entirely comprising instances of a single organism. Monocultures are rare in nature, and for good reason: they risk extinction from pathogens and have less chance of adapting to changing conditions. A pathogen could destroy some members of a diverse population but not all of them—diversity thus helps ensure survival of the population.

Although nature abhors monocultures, cyberspace seems to favor them. A collection of identical computing platforms is easier, hence cheaper, to manage because mastering one interface and making one set of configuration decisions suffices for all. In addition, user training costs are reduced when job transfers do not have the overhead of learning yet another operating system and suite of applications; investments in education about how to use or manage a system also can be amortized over a larger user base in a monoculture. Finally, interoperability of a few different kinds of systems is far easier to orchestrate than integrating a diverse collection, standards not withstanding. So networking is usually easier to support within a monoculture.

Mindful of these advantages, the public and private sectors both tend to adopt procurement policies that foster creating computer monocultures. The past five decades of computer usage in organizations has been a series of epochs, each one characterized by a single dominant instruction set architecture and operating system. Today it is Intel's x86 architecture running Microsoft's software.

Two things are different today than in the past, though:

- the widespread dependence on computing systems for day-to-day operations, and
- the interconnection of computing systems, which enables computers to exchange content (including malware).

These trends are somewhat incompatible. The first implies that an organization's computing infrastructure must be trustworthy for that organization to survive; the second means malware has an efficient way to attack, propagate, and compromise all members of the organization's computing infrastructure. The prospect of a computer monoculture thus terrifies computer security experts.

This terror is senseless. We argue in this article that a monoculture might well be a good cyberdefense strategy—at least for today. We also outline the kinds of attacks that likely will be launched when a monoculture defense is put in place, and we discuss what must be done to defend against them. Our analysis is holistic, based on how defenses and attacks are likely to coevolve. Although viewing the landscape in terms of the attacker reactions evoked by successive generations of defenses is unusual, we found it an enlightening exercise and believe it might well be a useful standard against which future defenses ought to be evaluated.

Vulnerabilities and Defenses

Different classes of attacks warrant different defenses. For the discussion that follows, we group attacks into three classes. (We have not tried to prove that these classes cover all possible attacks or that they actually constitute a partition on the space. The analysis in this article, however, depends on neither.) A *configuration attack* exploits a vulnerability introduced by the vendor-supplied default configuration, system administrator, or user who configures the software. Modern software systems are quite flexible, employing configuration files and global databases to customize each installation. Whether this customization is automated or manual, misconfiguration is a common source of vulnerabilities. Moreover, even when customization is not undertaken, vendor-supplied default configuration files historically have all too often permitted improper access to privileged functionality.

A *technology attack* exploits programming or design errors in software running on the target. All large systems have bugs, a situation that is not likely to change anytime soon. Inadequate specifications are also a serious problem, so even software that does what it was designed to do could have unintended side effects attackers can exploit. Thus, large systems invariably are vulnerable to technology attacks. Choosing the programming language wisely and using other software engineering tools can help software developers to eliminate some vulnerabilities, but the full spectrum of software issues is unlikely to yield to any technique known or even on the horizon.

Networked systems admit the possibility of *trust attacks*. In them, one computer satisfies a request from another because it trusts the source of the request, but in fact the source has already been compromised by an attacker. Of particular concern in the world of "cloud computing" is the tendency to group networked computers into enclaves, in which requests from within the enclave are deemed more trustworthy than those from outside. Once the attacker has compromised any computer in the enclave, the entire enclave is potentially at risk. Trust in the network itself is also a serious problem. Today, routing and address-mapping in the Internet are easy to compromise, Web pages can and are modified en route, and even the act of rendering a Web page can place a client system at risk due to the growing prevalence of scripting.

Defending Against Configuration Attacks

Configuration errors are an overwhelming source of vulnerability in today's systems and are particularly easy to exploit. Deploying a monoculture helps defend against such attacks because a single locked-down, well-understood configuration will have fewer vulnerabilities by virtue of the care invested in constructing that configuration. Even when systems are complex and configurations are unavoidably location- and user-specific, deploying a limited number of preanalyzed configurations might suffice to cover most needs without exposing known vulnerabilities.

In contrast, deploying a highly diverse system entails configuring each platform separately and ensuring that all of these different configurations are mutually compatible. Such an undertaking is an error-prone process.[1] Our conclusion is that if you believe configuration errors are a significant vulnerability today, then devoting the effort to eliminate configuration errors and then switching to a monoculture can be a cost-effective defense. However, if this course is pursued but configuration errors remain, then the payoff from a successful attack can be considerable.

Defending Against Technology Attacks

Reduce the opportunities for configuration attacks by deploying a monoculture of carefully analyzed configurations, and attackers will pursue technology attacks; thus, defenders must be prepared for that eventuality.

Defending a monoculture against technology attacks raises two separate issues. The first concerns defending against technology attacks per se on each platform—this depends only on the platform and not on the networked system in which it operates. The second issue is to increase the work an adversary requires to develop and launch technology attacks that spread rapidly and compromise a significant fraction of the individual platforms that make up the networked system. Monocultures benefit attackers here to the extent that attacks succeeding on one platform are likely to succeed on all.

A defense that addresses both issues is to use tools that automatically introduce diversity into the code executed on individual platforms. Various approaches have been proposed, including: relocation or padding the runtime stack by random amounts,[2–4] rearranging basic blocks and code within basic blocks,[2] randomly changing the names of system calls[5] or instruction opcodes,[6–8] and randomizing the heap memory allocator.[9] Some of these forms of *artificial diversity* are highly effective; others somewhat less so. For example, Hovav Shacham and colleagues derive experimental limits on the address space randomization scheme[10] that Jun Xu and colleagues proposed,[4] while Ana Sovarel and colleagues' work[11] discusses the effectiveness of instruction set randomization and outlines some attacks against it.

In all cases, artificial diversity defends against attacks by changing aspects of the implementation in ways that force attackers to individualize exploits. With code or storage layout no longer easily predictable, executing an attack is likely to raise a runtime error after a small number of instructions. So attacks that seek to compromise integrity or confidentiality will not succeed; the inputs will either be rejected or, in the worst case, cause the platform to crash. The defense, however, is probabilistic with respect to any given individual platform. First, an attack might be wholly unaffected by changes that artificial diversity introduces, because the attack does not depend on the changed implementation details at the target platform. (Certain forms of artificial diversity, though, affect just about all the software that executes on a platform—randomizing the names of system calls or instruction opcodes, for example.) Second, even attacks that fail, if repeated with enough different variants or if they yield detailed knowledge of the executable running on some target, might tell an attacker enough to craft an attack that works; the determined attacker can thus expend effort and increase the chances of success (the usual trade-off for a defense).

Note that artificial diversity does not protect against *interface attacks*, which involve exploiting desired functionality in unintended ways. Attacks packaged as scripts to be executed by an interpreter are prominent examples. The interface to the interpreter cannot be changed because scripts sometimes come from outside of the organization. And adding artificial diversity to the implementation of the interpreter does not change the effects of executing a script, hence does not defend against scripts that contain attacks.

By converting some attacks into crashes, artificial diversity can adversely affect a system's availability. Some systems will tolerate transient outages of individual platforms (perhaps recovering from crashes by running yet a different version of the implementation), but even systems that use replication to mask outages are limited by a fixed number of replicas. Thus, there is some probability that an attack might cause too many of the individual platforms that constitute a system to crash, thereby compromising the system's availability. The shape of such probability distributions is not well understood; they depend on the space and probability of various attacks, as well as the kind of artificial diversity.

Beyond defending individual platforms and systems, artificial diversity serves as an antidote to a monoculture's vulnerabilities. A platform that crashes in response to any attack cannot then help propagate that attack to other platforms and signal to system operators that something is wrong, thereby inviting the use of other means (which might well be out of band) to prevent the spread of whatever malware is serving as the attack vector. So, the spread of attack vectors that monocultures otherwise enable is slowed by artificial diversity. And this defense works in a manner complementary to other defenses for blocking the spread of malware through technology attacks.

One hesitation software developers voice about artificial diversity involves testing and debugging. With each deployed system having different internals, testing can now cover only a small fraction of what gets fielded. Moreover, when a system does crash, dumps and other diagnostic information must be interpreted in light of the diversity now present in the specific platform, which requires somewhat more sophisticated debugging and monitoring tools. These problems are far from insurmountable given modern programming environments. Microsoft's Vista, for example, is a widely deployed operating system that supports address-space randomization.

Critics of software monocultures advocate using true diversity for slowing the spread of malware perpetrating a technology attack. Different interfaces and operations having different semantics means true diversity can sometimes prevent interface attacks, whereas artificial diversity never can. However, with different interfaces and functionality, individual systems could in aggregate exhibit more different vulnerabilities, which helps attackers. Moreover, the cost of building (or acquiring) many different instances of the same kind of system is likely to be prohibitive for a system with thousands of workstations, as found in a moderately sized organization. And simply having independent teams build separate systems from the same specification does not preclude these systems from having identical vulnerabilities—for example, all teams might misinterpret a confusing specification in the same way. Finally, with true diversity, we again face the prospect of different configurations, so we lose one of the benefits of a monoculture.

Defending Against Trust Attacks

Diversity, whether artificial or true, multiplies the number of distinct attacks that can compromise some platform someplace in the system. An attack that fails at one platform might succeed at another. So instead of seeking an attack for a particular platform instance, an attacker could flood a network or individually probe all platform instances with a single attack. Some instance might succumb. If one does, and if other platforms are vulnerable to trust attacks, then the attacker can compromise those other platforms as well.

Thus, after deploying a monoculture to defend against configuration attacks and employing artificial diversity to help resist technology attacks, we should institute defenses against trust attacks. One obvious solution is to revisit the practice of decomposing networked systems into enclaves in which sites within an enclave trust each other more than they trust sites outside of the enclave. Another solution, long advocated but difficult to manage in practice, would be to employ fine-grained least-privilege authorization policies so that the operations one site performs on behalf of another are limited in scope and consequence.

With only finite resources, you should focus on only those threats perceived to be real. Knowledge of the threats—including resources available to them, likely expertise, and probable goals—helps identify the targets that must be defended and predict what kinds of attacks are plausible. Our analysis in this article is predicated on a presumption that the low-hanging fruit for attackers today is configuration attacks. It would be nice to have that assumption validated, but even without that validation, our arguments clearly show that it is naive to regard deploying a monoculture as a risk that cannot be mitigated. On the contrary, we find many reasons to believe that a monoculture could be made far more robust than what it likely replaces.

The deployment of a monoculture should be viewed in the context of how it affects the evolution of attacker responses to defenses. Whereas defenders today cannot hope to defend against all attacks, they can deploy defenses with an eye toward anticipating the vulnerabilities new generations of attacks will exploit. A monoculture defends against some attacks (configuration attacks) but creates new vulnerabilities to technology attacks; employing artificial diversity in this monoculture defends against some of those technology attacks but could increase vulnerability to trust attacks; and so on.

The characterization of monoculture in this article is particularly well suited for understanding the effects of procurement policies that restrict computer platform acquisitions to systems from a single vendor running a standard configuration. This, however, is not the only way in which a monoculture might arise. Any standard will create a kind of monoculture—namely, the ubiquitous deployment of interfaces and services implementing that standard. And the more widely adopted the standard, the greater the incentive for developing attacks. For example, Web services will admit technology attacks that not only involve exploiting the semantics of system internals but also might involve the interfaces themselves. The diversity defense is not currently an option for defending against attacks that exploit the misguided semantics of an interface. □

Acknowledgments

We are grateful to C. Chandersekaran, Jay Lala, Butler Lampson, John Manferdelli, Gene Spafford, and Vijay Varadharajan for their thoughts on monocultures topic and to the participants of the AFOSR workshop on Homogeneous Enclave Software vs. Controlled Heterogeneous Enclave Software. We also thank two anonymous reviewers. The authors are supported in part by AFOSR grant F9550-06-0019, AFOSR grant FA9550-07-1-0569, and US National Science Foundation grants 0430161 and CCF-0424422 (TRUST).

References

1. D. Oppenheimer, A. Ganapathi, and D.A. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" *Proc. 4th Usenix Symp. Internet Technologies and Systems*, Usenix Assoc., 2003, pp. 1–16.
2. S. Forrest, A. Somayaji, and D.H. Ackley, "Building Diverse Computer Systems," *Proc. 6th Workshop Hot Topics in Operating Systems*, IEEE CS Press, 1997, pp. 67–72.
3. S. Bhatkar, D.C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," *Proc. 12th Usenix Security Symp.*, Usenix Assoc., 2003, pp. 105–120.
4. J. Xu, Z. Kalbarczyk, and R.K. Iyer, "Transparent Runtime Randomization for Security," *Proc. 22nd Int'l Symp. Reliable Distributed Systems* (SRDS 03), IEEE CS Press, 2003, pp. 260–269.
5. M. Chew and D. Song, *Mitigating Buffer Overflows by Operating System Randomization*, tech. report CMU-CS-02-197, School of Computer Science, Carnegie Mellon Univ., 2002.
6. G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," *Proc. 10th ACM Conf. Computer and Communications Security* (CCS 03), ACM Press, 2003, pp. 272–280.
7. E.G. Barrantes et al., "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," *Proc. 10th ACM Conf. Computer and Communications Security* (CCS 03), ACM Press, 2003, pp. 281–289.
8. E.G. Barrantes et al., "Randomized Instruction Set Emulation," *ACM Trans. Information and System Security*, vol. 8, no. 1, 2005, pp. 3–40.
9. E.D. Berger and B.G. Zorn, *DieHard: Probabilistic Memory Safety for Unsafe Languages*, tech. report 05-65, Dept. of Computer Science, Univ. of Massachusetts Amherst, 2005.
10. H. Shacham et al., "On the Effectiveness of Address-Space Randomization," *Proc. 11th ACM Conf. Computer and Communications Security* (CCS 04), ACM Press, 2004, pp. 298–307.
11. A.N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB?: The Effectiveness of Instruction Set Randomization," *Proc. 14th Usenix Security Symp.*, Usenix Assoc., 2005, pp. 145–160.

Kenneth P. Birman *is a professor at Cornell University's computer science department. His research interests focus on challenges of scalability, fault tolerance, and consistency in distributed systems. Birman is probably best known for developing the virtual synchrony replication model and the Isis Toolkit, the first system to support that model. He has a PhD in computer science from the University of California, Berkeley. He is a fellow of the ACM and the author of three textbooks, most recently* Reliable Distributed Systems: Technologies, Web Services, and Applications *(Springer Verlag, 1996). Contact him at ken@cs.cornell.edu.*

Fred B. Schneider *is a professor at Cornell University's computer science department and chief scientist for the multi-university TRUST NSF-funded Science and Technology Center. A more detailed biography appears on p. 13.*