Building Diverse Computer Systems

Stephanie Forrest[†] steph@ai.mit.edu
Anil Somayaji[†] soma@ai.mit.edu
David H. Ackley ackley@cs.unm.edu

Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131

Submitted to the Sixth Workshop on Hot Topics in Operating Systems
January 16, 1997

1 Introduction: Diversity is valuable

In biological systems, diversity is an important source of robustness. A stable ecosystem, for example, contains many different species which occur in highly-conserved frequency distributions. If this diversity is lost and a few species become dominant, the ecosystem becomes susceptible to perturbation s such as catastrophic fires, infestations, and disease. Similarly, health problems often emerge when there is low genetic diversity within a species, as in the case of endangered species or animal breeding programs. The vertebrate immune system offers a third example, providing each individual with a unique set of immunological defenses, helping to control the spread of disease within a population.

Computers, by contrast, are notable for their lack of diversity. Manufacturers produce multitudes of identical copies from a single design, with the goal of making every chip of a given type and every copy of a given program identical. Beyond the economic leverage provided by the massive cloning of a single design, such homogeneous systems have other advantages: They behave consistently, application software is more portable and more likely to run identically across machines, debugging is simplified, and distribution and maintenance tasks are eased. Standardization efforts reflect the almost universal belief that homogeneity is beneficial.

As computers increasingly become mass-market commodities, the decline in the diversity of available hardware and software is likely to continue, and as in biological systems, such a development carries serious risks. All the advantages of uniformity become potential weaknesses when they can be exploited by an attacker, because once a method is created for penetrating the security of one computer, all computers with the same configuration become similarly vulnerable. The potential danger grows with the population of interconnected and homogeneous computers.

In this paper, we argue that the beneficial effects of diversity in computing systems have been overlooked, and we discuss methods by which diversity could be enhanced with minimal impact on convenience, usability, and efficiency. Although diversity considerations affect computing at many levels, here we focus primarily on computer security, and our emphasis is on diversity at the software level, particularly for operating systems, which are a common point of intrusion.

Computer security is a growing concern for open computing environments. Malicious intrusions are multiplying as huge numbers of people connect to the Internet, exchange electronic mail and

[†]Current address: MIT AI Laboratory, 545 Technology Sq., Cambridge, MA 02139.

commercially valuable data, download files, and run computer programs remotely, often across international boundaries. Traditional approaches to computer security—based on passwords, access controls, and so forth—are ineffective when an attacker is able to bypass them by exploiting some unintended property of a system. Finding ways to mitigate such attacks is likely to be an increasing concern for the operating systems community.

Deliberately introducing diversity into computer systems can make them more robust to easily replicated attacks. More speculatively, it might also enhance early detection of timing problems and other bugs. Today, each new discovery of a security hole in any operating system is a serious problem, because all of the installed base of that operating system—thousands, if not millions, of machines, running almost exactly the same system software—may well be vulnerable. An attack script developed on one machine is likely to work on thousands of other machines. If every intrusion, virus, or worm had to be explicitly crafted to a particular machine, the cost of trying to penetrate computer systems would go up dramatically. Only sites with high-value information would be worth attacking, and these could be protected using other methods. The relevance of diversity to computer systems was recognized as early as 1989 in the aftermath of the Morris Worm [2], when it was observed that only a few machine types were vulnerable to infection. Yet, this simple principle has not been adopted in any computer security system that we know of.

2 Strategy: Avoid unnecessary consistency

Our goal is to prevent widespread attacks by making intrusions much harder to replicate. Can we introduce diversity in a way that will tend to disrupt malicious attacks—even through security holes that haven't been discovered yet—without compromising reliability, efficiency, and convenience for legitimate users? We believe that the answer is yes, because computer systems today are far more consistent than necessary. For example, all but the lowest-level computational tasks are now implemented in a high-level programming language, and for every such program there are many different translations into machine code that will accomplish the same task. Each aspect of a programming language that is "arbitrary" or "implementation dependent" is an opportunity for *randomized compilation* techniques to introduce diversity. Such diversity would preserve the functionality of well-behaved programs (we refer to this as "the box") and be highly likely to disrupt others by removing unnecessary regularities ("surrounding the box with noise").

We have adopted the following guidelines to help us identify the most promising directions to explore:

1. Preserve high-level functionality. At the user level, the behavior of different systems should be predictable, and the input/output behavior of programs should be identical on different computers.

2. Introduce diversity in places that will be most disruptive to known intrusion methods. However, documenting the most common routes of intrusion is difficult for several reasons: (a) new routes of intrusion are continually being discovered, (b) old routes of intrusion are sometimes patched, (c) there are few if any reliable statistics on successful intrusions, and (d) there is a distinction between the variety of intrusion methods and the frequency with which they are exploited.

3. Minimize costs, both run-time performance and the cost of introducing and maintaining diversity. We believe that the latter is likely be directly related to where the variations are

introduced in the software development process [6]. A load-time modification is likely to be less costly than a compile-time modification which in turn is less costly than requiring a developer to write multiple versions of application code.

4. Introduce diversity through randomization. Randomization methods are likely to scale well.

3 Possible Implementations

There are a wide variety of possible implementation strategies for introducing diversity. In this section, we discuss several of these and their implications for security. Our emphasis is on variability that can be introduced into software sometime between the time that the software is developed and when it is executed. The methods range from those that produce variability in the physical location of executed instructions, the order in which instructions are executed, the location of instructions in memory at run-time, and the ability of running code to access external routines, files, and other resources:

1. No-Ops: Perhaps the simplest method is to insert no-ops or other nonfunctional code in compiled code at random locations. Depending on the architecture, this could potentially affect timing relations in running code and would slightly change the physical location of instructions. The timing attacks reported on RSA [3] could potentially be disrupted using this method, although other remedies have also been proposed.
2. Reordering code: Optimizing and parallelizing compilers use many techniques to improve performance, and some of these could be used to generate code variations. For example,
    (a) Basic blocks: Rearrange the basic blocks of compiled code in random order [1]. This would cause instructions to be stored in different locations but would not affect the order in which they are executed, although there would likely be some long jumps executed instead of short jumps. This method could potentially disrupt some viruses. However, most file-infector viruses insert a single jump instruction that transfers control to the virus code (stored at the end of the program), and then return control to the original program. Thus, rearranging basic blocks in the program segment would be unlikely to affect this large class of viruses.
    (b) Optimizations for parallel processing: Many techniques already exist for producing blocks of instructions that can be run simultaneously on multiple processors. These techniques could be applied to code intended for execution on a single processor, resulting in code that is executed in a unique order. We don't know what if any intrusion methods this would disrupt. Further, the amount of variability that could be produced with this method would be limited to the amount of parallelism that could be extracted from the original program.
    (c) Vary the order of instructions within a basic block, while respecting the data and control dependencies present in the source code. A preliminary study of the source code for the Linux kernel concluded that the number of different orderings that could be automatically generated was very high [5].
3. Memory layout: There are standard ways of allocating memory when code runs and of ordering the components of memory. These are arbitrary and could be varied in many ways. Here are a few examples:

   (a) Pad each stack frame by a random amount (so the return addresses are not located in predictable locations).

   (b) Randomize the locations of global variables, and the offsets assigned to local variables within a stack frame.

   (c) Assign each newly allocated stack frame in an unpredictable (e.g., randomly chosen) location instead of in the next contiguous location). This would amount to treating the stack as a free store, which would result in some memory-management overhead.

   Some of these memory-layout schemes would likely disrupt a pervasive form of attack—the buffer overflow. There are several potential complications, however, including whether and how to preserve ABI compatibility, debuggers, how to preserve the correct functionality for the C function "alloca," and how maintain compatibility with dynamic libraries.

4. Process initialization: Code that runs before user code executes could be varied, an example being the startup files that give information about how to load a process image into virtual memory [8].

5. Dynamic libraries: It should be possible to vary the way in which libraries are accessed, the locations in which they are stored, or the information that they contain for each different system. If every routine used by the system is stored in a unique location, or has a unique naming scheme (e.g., by varying the system call numbering), then the code that accesses those routines can only run if it knows the correct locations (or numbers). This idea is appealing because it could be implemented at the shared library dynamic loader level (close to run-time), but we don't yet have a good analysis of what advantages it would confer in terms of security.

6. Random/unique names for system files: Varying the locations of common system files so they are difficult for intruding code to find would be highly effective against a wide range of attacks. This would also tend to complicate system administration, however, and therefore is unlikely to be acceptable at this time.

7. Magic numbers in certain files, e.g., executables: The type of information contained in many files can be (at least tentatively) identified by searching for characteristic signatures at the beginning of the file. Individual systems could re-map such signatures to randomly-chosen alternatives and convert the signatures of externally-obtained files via an explicit "importation" process.

8. Randomized run-time checks: Many successful intrusions could be prevented if all compiled code performed dynamic array bounds checking [4]. However, such checks are rarely performed in production code because of perceived performance costs. Instead of requiring every program to pay the cost of doing complete dynamic checking, each executing program could perform some of these checks (potentially a very small number of them). Which checks were to be performed could be determined statically (at compile time) or dynamically (at run-time).

4 Preliminary Results

As an initial demonstration of these ideas, we have implemented a simple method for randomizing the amount of memory allocated on a stack frame and shown that it disrupts a simple buffer overflow attack (idea 3a from the previous section). Buffer overflow attacks arise because many programs statically allocate storage for input, and then do not ensure that their received input fits within the allotted space. Because C does not automatically check array bounds, overflows can result in the corruption of adjacent variables and/or code. Buffer overflows are problematic in the context of programs that run as root in UNIX, primarily because they provide a way for a non-privileged user to obtain root access. However, any script exploiting such vulnerabilities is brittle. In order to overwrite the return address, the size of the buffer and the relative location of the function's return address on the stack must be known. Further, in order to execute arbitrary code inserted into the buffer, the exact location of the buffer in memory must also be known.

If every compilation produced an executable with a different stack layout, then exploit scripts developed on one executable would have a low probability of success on other executables. To change the layout of the stack, we increase the size of the stack frame by a random amount, simply by adding a fixed amount of space to randomly selected stack slots. Such additions affect both the stack layout for the modified function and the exact locations of every function called by it. Such a change will of course cause a program to expand its use of stack memory, leading to some increased cache miss rates and other performance penalties. To implement this, we made a small modification to `gcc`, so that it adds four bytes to a stack slot, based on a biased coin flip. For a more complete description, see [7].

The revised version of `gcc` produces a program that disrupts a simple buffer overflow attack. An important question is how much extra stack space is required for this method to be effective. We assumed that a 10% increase in stack space would be permissible, and we then studied some common UNIX programs to see how much variability we could achieve under this constraint. For the three programs we examined (`sendmail-8.8.4`, `wu-ftpd-2.4`, `apache_1.2b1`), the number of possible variants is large. The smallest, `wu-ftpd`, has 2^{274} possible variants and approximately $2^{\binom{274}{27}}$ variants within the 10% memory constraint. Thus, stack space can be traded off against security, simply by setting the slot padding probability. This gives one example of how a compiler could help users create unique systems, which are vulnerable to attack but vulnerable in ways different from every other computer.

5 Impact on Computer Security

Here we give a brief overview of common security problems and our assessment of which diversity methods would be most effective against them. Code bugs (e.g., buffer overflows, insecurely processing command-line options, symlink errors, temp file problems, etc.) constitute a common form of attack. Memory layout variations, such as the one we implemented, would address several of these. A race condition is an interaction between two normally operating programs via some shared resource (often, a file). Compilation techniques are unlikely to prevent race conditions, but diversity at the level of the shared resource would likely be effective. For configuration problems (e.g., setup errors in how a service is provided or file permission problems), unique naming of system files would be highly effective. Denial-of-service attacks are sometimes due to code bugs and sometimes due to lack of resource checking or poor policies. Thus, one diversity technique alone is unlikely to address all denial-of-service problems. For problems associated with insecure channels (e.g., IP spoofing, terminal hijacking, etc.), we expect that cryptography techniques are probably more helpful than diversity techniques, at least for diversity generated on a single host. Trust abuse, including key management problems and inappropriately trusted IP addresses, could be addressed by generating a unique profile of each computer's behavior and using it to establish identity. A final security problem that has been well-studied is that of covert channels. It might be possible to introduce diversity to prevent exploitation of covert channels, but we haven't thought about it carefully.

6 Conclusion

Diversity techniques such as those we have proposed here can serve an important role in the development of more robust and secure computing systems. They cannot, by themselves, solve all security problems, because many exploitable holes are created completely "within the box" of a program functioning under the semantics of the language in which it is written. And indeed, diversity techniques may sometimes disrupt legitimate use by unmasking unintended implementation dependencies (i.e., "bugs") in benign code. Nonetheless, the essential principles of diversity—"avoid unnecessary consistency," and "surround the box with noise"—express a strategy that is likely to find use in the computers of the future.

Acknowledgments

Over the past couple of years we have discussed the general idea of diversity with many people and solicited their comments. In particular, A. Davis, T. Knight, M. Oprea, B. Maccabe, M. Seltzer, H. Shrobe, and E. Stoltz have all listened with more or less open minds and made helpful suggestions. Where possible, we have acknowledged their individual contributions. The authors gratefully acknowledge support from the National Science Foundation (grant IRI-9157644), the Office of Naval Research (grant N00014-95-1-0364), Defense Advanced Research Projects Agency (grants N00014-96-1-0680 and N66001-96-C-8509), the MIT AI Lab., Interval Research Corp., and the Santa Fe Institute.

References

[1] A. L. Davis, 1996. personal communication.

[2] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of november 1988. In *Proceedings of the IEEE Symposium on Research in Computer Security and Privacy*, Los Alamitos, CA, 1989. IEEE, IEEE Computer Society Press.

[3] E. English and S. Hamilton. Network security under siege: the timing attack. *Computer*, March 1996.

[4] T. Knight, 1996. personal communication.

[5] M. Oprea. Towards compiler-induced object code variability. Unpublished Manuscript, June 1996.

[6] H. Shrobe, 1996. personal communication.

[7] A. Somayaji. Object-code variation for enhanced security. Unpublished Manuscript, December 1996.

[8] E. Stoltz, 1996. personal communication.