Attacking Malicious Code: A Report to the Infosec Research Council

The accelerating trends of interconnectedness, complexity, and extensibility are aggravating the already-serious threat posed by malicious code. To combat malicious code, these authors argue for creating sound policy about software behavior and enforcing that policy through technological means.

Gary McGraw, *Cigital*
Greg Morrisett, *Cornell University*

In October of 1999, the Infosec Research Council created a Science and Technology Study Group focused on malicious code. The Malicious Code ISTSG is charged with developing a national research agenda to address the accelerating threat from malicious code. The study is intended to identify promising new approaches to dealing with the problems posed by malicious code. In this report, we discuss important trends that are making malicious code an increasingly serious problem. We then survey existing techniques for preventing attacks, pointing out their limitations, and discuss some promising new approaches that might address these limitations.

This report is a byproduct of two meetings of Study Group members and their invited guests. Although this report was written by two of the study group members, we believe it represents an accurate distillation of the ideas and insights of all the participants.

What is Malicious Code?

Malicious code is any code added, changed, or removed from a software system to intentionally cause harm or subvert the system's intended function. Although the problem of malicious code has a long history, a number of recent, widely publicized attacks and certain economic trends suggest that malicious code is rapidly becoming a critical problem for industry, government, and individuals.

Traditional examples of malicious code include viruses, worms, Trojan Horses, and attack scripts, while more modern examples include Java attack applets and dangerous ActiveX controls:

- Viruses are pieces of malicious code that attach to host programs and propagate when an infected program executes.
- Worms are particular to networked

Extensible systems, including computers, are particularly susceptible to the malicious functionality problem.

computers. Instead of attaching themselves to a host program, worms carry out programmed attacks to jump from machine to machine across the network.
- Trojan Horses, like viruses, hide malicious intent inside a host program that appears to do something useful (such as a program that captures passwords by masquerading as the login daemon).
- Attack scripts are programs written by experts that exploit security weaknesses, usually across the network, to carry out an attack. Attack scripts exploiting buffer overflows by "smashing the stack" are the most commonly encountered variety.
- Java attack applets are programs embedded in Web pages that achieve foothold through a Web browser.
- Dangerous ActiveX controls are program components that allow a malicious code fragment to control applications or the operating system.

Recently, the distinctions between malicious code categories have been bleeding together, making classification difficult. Table 1 provides some concrete examples of malicious code. Recent versions of malicious code are really amalgamations of different categories.

A Growing Problem

Complex devices, by their very nature, introduce the risk that malicious functionality can be added (either during creation or afterwards) that extends the original device past its primary intended design. As an unfortunate side effect, inherent complexity lets malicious subsystems remain invisible to unsuspecting users until it is too late. Some of the earliest malicious functionality, for example, was associated with complicated copy machines. Extensible systems, including computers, are particularly susceptible to the malicious functionality problem. When extending a system is as easy as writing and installing a program, the risk of intentional introduction of malicious behavior increases drastically.

Any computing system is susceptible to malicious code. Rogue programmers can modify systems software that is initially installed on the machine. Users might unwittingly propagate a virus by installing new programs or software updates from a CDROM. In a multi-user system, a hostile user might install a Trojan Horse to collect other users' passwords. These attack vectors have been well known since the dawn of computing, so why is malicious code a bigger problem now than in the past? We argue that a small number of trends have a large influence on the recent widespread propagation of malicious code.

Networks Are Everywhere

The growing connectivity of computers through the Internet has increased both the number of attack vectors and the ease with which an attack can be made. More and more computers, ranging from home PCs to systems that control critical infrastructures (such as the power grid), are being connected to the Internet. Furthermore, people, businesses, and governments are increasingly dependent upon network-enabled communication such as e-mail or Web pages provided by information systems. Unfortunately, as these systems are connected to the Internet, they become vulnerable to attacks from distant sources. Put simply, an attacker no longer needs physical access to a system to install or propagate malicious code.

Because access through a network does not require human intervention, launching automated attacks from the comfort of your living room is relatively easy. Indeed, the recent denial-of-service attacks in February of 2000 took advantage of a number of (previously compromised) hosts to flood popular e-commerce Web sites with bogus requests automatically. The ubiquity of networking means that there are more systems to attack, more attacks, and greater risks from malicious code than in the past.

System Complexity Is Rising

A second trend that has enabled widespread propagation of malicious code is the size and complexity of modern information systems. A desktop system running Windows/NT and associated applications depends upon the proper functioning of the kernel as well as the applications to ensure that malicious code cannot corrupt the system. However, NT itself consists of tens of millions of lines of code, and applications are becoming equally, if not more, complex. When systems become this large, bugs cannot be avoided. Exacerbating this problem is the use of unsafe programming languages

Table 1

Examples of Malicious Code

Malicious code	Date	Category	Explanation
Love Bug	2000	Mobile code virus	The fastest spreading virus of all time used VB script and Microsoft Outlook mail to propagate. Caused an estimated $10 billion in damage.
Trinoo (and other dDoS scripts)	2000	Remote-control attack script	The highly publicized denial of service attacks of February 2000 were carried out by remotely planted agent programs.
Melissa	1999	Mobile code virus	The second fastest spreading virus of all time used e-mail to propagate. Infected over 1.2 million machines in a few hours.
Explore.Zip	1999	Mobile code worm	An e-mail borne worm that exploited problems in Microsoft Windows to propagate.
Happy99	1999	Virus	A widespread virus infecting Microsoft PCs.
CIH	1998	Virus	A particularly dangerous virus that attacks BIOS in PCs. Ran rampant in Asia before contained.
Back Orifice	1998	Offensive code	Remote control program installed on Windows machines by crackers. Pervasive.
Attack scripts		Offensive code	Crackers called "script kiddies" download malicious code from the Internet and run it against any number of targets. Some expert must create and release the script to begin with. Widespread. Most common attack: buffer overflow.
ActiveX (scripting)	1997	Mobile code	Decried by security professionals, Microsoft's ActiveX system introduces grave security risks by relying on user's discretion and judgment even though digital signatures are used.
Java Attack Applets	1996-2000	Mobile code	Attack applets placed on Web sites take advantage of flaws in the Java security model to carry out attacks. 19 known attacks.
Morris worm	1988	Worm	Released in 1988 by Robert Morris, Jr, this program affected around 6000 computers (around 10% of the Internet at the time).
Thompson's compiler trick	1984	Trojan Horse	Ken Thompson introduced a Trojan Horse in a C compiler that compiled itself into future programs ("Reflections on Trusting Trust," *Comm. ACM*, Vol. 27, No. 8, Aug. 1984).

(such as C or C++) that do not protect against simple kinds of attacks, such as buffer overflows. However, even if the systems and applications code were bug free, improper configuration by retailers, administrators, or users can open the door to malicious code. In addition to providing more avenues for attack, complex systems make it easier to hide or mask malicious code. In theory, we could analyze and prove that a small program was free of malicious code, but this task is impossible for even the simplest desktop systems today, much less the enterprise-wide systems used by businesses or governments.

Systems Are Easily Extensible

A third trend enabling malicious code is the degree to which systems have become extensible. An extensible host accepts updates or extensions, sometimes referred to as *mobile code*, so that the system's functionality can be evolved in an incremental fashion. For example, the plug-in architecture of Web browsers makes it easy to install viewer extensions for new document types as needed. Today's operating systems support extensibility through dynamically loadable device drivers and modules. Today's applications, such as word processors, e-mail clients, spreadsheets, and Web browsers, support extensibility through scripting, controls, components, and applets. From an economic standpoint, extensible systems are attractive because they provide flexible interfaces that can be adapted through new components. In today's marketplace, it is crucial that software be deployed as rapidly as possible to gain market share. Yet the marketplace also demands that applications provide new features with each release. An extensible architecture makes it easy to satisfy both demands by letting companies ship the base application code early and later ship feature extensions as needed.

Unfortunately, the very nature of extensible systems makes it hard to prevent malicious code from slipping in as an unwanted extension. For example, the Melissa virus took advantage of the scripting extensions of Microsoft's Outlook e-mail client to propagate itself. The virus was coded as a script contained in what appeared to users as an innocuous mail message. When users opened the message, the script executed, proceeded to obtain email addresses from the user's contacts database, and then sent copies of itself to those addresses. The infamous Love Bug worked very similarly, also taking advantage of Outlook's scripting capabilities.

Defense against Malicious Code

Creating malicious code is not hard. In fact, it is as simple as writing a program or

To make matters worse, our traditional tools for ensuring the security and integrity of hosts have not kept pace with the ever-changing suite of applications.

downloading and configuring a set of easily customized components. It is becoming increasingly easy to hide ill-intentioned code inside otherwise innocuous objects, including Web pages and e-mail messages. This makes detecting and stopping malicious code before it can do any damage extremely hard.

To make matters worse, our traditional tools for ensuring the security and integrity of hosts have not kept pace with the ever-changing suite of applications. For example, traditional security mechanisms for access control reside within an operating system kernel and protect relatively primitive objects (such as files); but increasingly, attacks such as the Melissa virus happen at the application level where the kernel has no opportunity to intervene.

A useful analogy is to think of today's computer and network security mechanisms as the walls, moats, and drawbridges of medieval times. At one point, these mechanisms were effective for defending our computing castles against isolated attacks, mounted on horseback. But the defenses have not kept pace with the attacks. Today, attackers have access to airplanes and laser-guided bombs that can easily bypass our antiquated defenses. In fact, attackers rarely need sophisticated equipment: because our kingdoms are really composed of hundreds of interconnected castles, attackers can easily move from site to site, finding places where we have left the drawbridge down. It is time to develop some new defenses.

In general, when a computational agent arrives at a host, there are four approaches that the host can take to protect itself:

- ***Analyze*** the code and reject it if there is the potential that executing it will cause harm.
- ***Rewrite*** the code before executing it so that it can do no harm.
- ***Monitor*** the code while its executing and stop it before it does harm, or
- ***Audit*** the code during executing and take policing action if it did some harm.

Code analysis includes simple techniques, such as scanning a file and rejecting it if contains any known virus, as well as more sophisticated techniques, including dataflow analysis, which can sometimes discover previously unseen malicious code. Analysis can also help locate security-related bugs (such as potential buffer overflow conditions) that malicious code can use to gain a foothold in a system. But analyses are necessarily limited, because determining if code will misbehave is as hard as the halting problem. Consequently, any analysis will either be too conservative (and reject some perfectly good code) or too permissive (and let some bad code in) or more likely, both. Furthermore, software engineers working on their own systems often neglect to apply any bug-finding analyses. Automated tools such as the open source security scanner ITS4 (see www.rstcorp.com/its4) and more sophisticated tools incorporating dataflow analysis can be effective for finding bugs.[1,2] In addition, primitive static analysis, such as looking for particular patterns of system calls in an executable, has been incorporated into some commercially available security products.

Code rewriting is a less pervasive approach to the problem, but might become more important (see the next section). With this approach, a rewriting tool inserts extra code to perform dynamic checks that ensure bad things cannot happen. For example, a Java compiler inserts code to check that each array index is in bounds—if not, the code throws an exception, thereby avoiding the common class of buffer overflow attacks. Rewriting can be carried out either at the application code level, or below that in subsystem functionality made available through APIs, or even at the binary level.

Monitoring programs, using a reference monitor, is the traditional approach used to ensure programs do not do anything bad. For instance, an operating system uses the page-translation hardware to monitor the set of addresses that an application attempts to read, write, or execute. If the application attempts to access memory outside of its address space, the kernel takes action (such as by signaling a segmentation fault.) A more recent example of an online reference monitor is the Java Virtual Machine interpreter. The interpreter monitors execution of applets and mediates access to system calls by examining the execution stack to determine who is issuing the system call request.[3] In this case, stack inspection serves as a policy enforcement mechanism.

If malicious code does damage, recovery is only possible if the damage can be properly assessed and addressed. Creating an audit trail that captures program behavior is an essential step. Several program-auditing tools are commercially available.

Each of the basic approaches—analysis, rewriting, monitoring, and auditing—has its strengths and weaknesses, but fortunately, these approaches are not mutually exclusive and can be used in concert. Of course, to employ any of them, we must first identify what could be "harmful" to a host. Like any other computing task, we must turn the vague idea of "harm" into a concrete, detailed specification—a security policy—so that it can be enforced by some automated security architecture. Therein lies our greatest danger, for as we create the policy, we are likely to abstract or forget relevant details of the system. An attacker will turn to these details first, stepping outside our policy model to circumvent the safeguards.

Stick to Your Principles

To protect against this common failing, it is important to follow well-established security principles when designing security policies. One of the most important principles, first stated by Jerome Saltzer and Michael Shroeder in 1975,[4] is the *Principle of Least Privilege:* a component should be given the minimum access necessary to accomplish its intended task. For example, we shouldn't give a program access to all files in a system but rather, only those files that the program needs to get its job done. This prevents the program from either accidentally or maliciously deleting or corrupting most files. Obviously, the fewer files that the program can access, the less the potential damage. Stated simply, tighter constraints on a program lead to better security.

Another important security principle is the *Principle of Minimum Trusted Computing Base*. The trusted computing base (TCB) is the set of hardware and software components that make up our security enforcement mechanisms. The Principle of Minimum TCB states that, in general, the best way to assure that your system is secure is to keep your TCB small and simple. Even in the mid 1970's, operating system kernels were thought to be too large to be trusted. Those systems now seem small and tightly structured compared to today's widely used kernels composed of millions of lines of code.

Current Defenses

We now turn to examples of currently deployed defenses for malicious code, focusing on their relative pros and cons. Unfortunately, the comparison shows that the pros are outweighed by the cons, largely because of a violation of the Least Privilege and Minimal TCB principles.

OS-Based Reference Monitors

Historically, mechanisms for security policy enforcement have been provided by the computer hardware and operating system. Address translation hardware, distinct supervisor- and user-modes, timer interrupts, and system calls for invoking a trusted software base serve in combination to enforce limited forms of availability, fault containment, and authorization properties.

To a large degree, these mechanisms have proven effective for protecting operating system resources (such as files or devices) from unauthorized access by humans or malicious code. But the mechanisms work with a fixed system-call interface and a fixed vocabulary of principals, objects, and operations for policies. Only by incurring significant cost and usability penalties can that vocabulary be expanded. It rarely is. Currently, most desktop machines are configured as single-user, so applications have complete access to the machine resources.

Scanning for Known Malicious Code

In the days before networking was rampant, malicious code mostly used the "sneaker net" as its vector. Viruses spread from machine to machine by humans carrying floppy disks with infected programs on them. Perhaps the built-in limitations in the vector kept the number of viruses small. In any case, the limited number of viruses combined with the inefficiencies in the communication vector made possible the strategy *of blacklisting*.

Blacklisting, a strategy used by most commercial antivirus products, matches programs against a database of known virus signatures (such as code fragments). If a match is found, the program is disabled. Today, commercial products scan not only binary programs, but also email messages, Web pages, or documents looking for viruses in the form of scripts. This approach's limitations are obvious. Unknown malicious code will easily get by the simple defenses to carry

In the days before networking was rampant, malicious code mostly used the "sneaker net" as its vector. Viruses spread from machine to machine by humans carrying floppy disks with infected programs on them.

Type systems, like software-based reference monitors, go beyond operating systems in that they can be used to enforce a wider class of application-specific access policies.

out its dirty work. Until vendors can contain a new virus and add a signature entry to the database, it can run rampant. Recall both the Melissa virus and the Love Bug. Another limitation of the approach is that it does not scale well; each file must be scanned against an ever growing list of viruses.

Clearly, blacklisting by itself does not provide adequate security. It is too easy to make trivial changes to malicious code (a process that can be automated in the code itself) to thwart almost every black listing scheme. Nevertheless, black listing is cheap to implement and is thus worthwhile even if it only stops the occasional naïve attack.

Code Signing

Code signing is an approach for authenticating code based on public-key cryptography and digital signatures. The digital signature lets a user determine which particular key the code was signed with and ensure (with high probability) that the code has not been tampered with since it was signed.

Unfortunately, most people assume that digital signatures imply a lot more than they really do. In particular, people typically assume that the signed code was signed by the owner of the key, that the owner of the key wrote the code, that the code is good, and that the code may be safely used in any context. But these assumptions are often not true!

For instance, if a key is stolen, anyone can use it to sign any piece of code—including malicious code. As another example, the developer might consider the code to be "good" and thus sign it, even if the code contains a Trojan horse or virus. Finally, what developers or retailers consider to be good might not be good for the user: A component that sends back information to the home office may seem useful to a vendor, but will probably be considered a violation of privacy by the user.

Thus, while code signing is a useful technology, it suffers from some real limitations not the least of which is poor understanding of what a digital signature really means. Furthermore, the adoption of code signing has been hampered by the lack of a Public Key Infrastructure. Very few PKI installations have been deployed, and those that have do not begin to approach Internet scale. Without a solid PKI, code signing will not become common.

Promising New Defenses

Now we'll discuss some promising technologies, identified by the study group, that are emerging from research labs.

Software-Based Reference Monitors

Robert Wahbe and his colleagues suggested *software-based fault isolation* as an alternative to the traditional hardware-based mechanisms used to ensure memory safety.[5] Their goal was to reduce the overhead of cross-domain procedure calls and prove a more flexible memory-safety mode. Their basic idea is to rewrite binary code by inserting checks on each memory access and each control transfer to ensure that those accesses are valid. Fred Schneider generalized the SFI idea to in-lined reference monitors.[6] With the IRM approach, a security policy is specified in a declarative language, and a general-purpose tool rewrites code, inserting extra checks and state that serve to enforce the policy. In principle, any security policy that is a safety property can be enforced, so the approach is quite powerful. For example, it can enforce any discretionary access control policy. The approach is also practical: Prototypes have been built at both Cornell and MIT.[7–9] One of the Cornell prototypes, PSLang/PoET, works for the Java Virtual Machine language and gives competitive performance for the implementation of Java's stack inspection security policy.

Type-Safe Languages

Type-safe programming languages, such as Java, Scheme, or ML, ensure that operations are only applied to values of the appropriate type. Type systems that support type abstraction let programmers specify new, abstract types and signatures for operations that prevent unauthorized code from applying the wrong operations to the wrong values. In this respect, type systems, like software-based reference monitors, go beyond operating systems in that they can be used to enforce a wider class of application-specific access policies. Static type systems also enable offline enforcement through static type checking instead of each time a particular operation is performed. This lets the type checker enforce certain policies that are difficult with online techniques. For example, Andrew Myers' Jflow extends the

Table 2

Examples of Malicious Code Understood in Our Policy-Based Framework

Bad policy	Examples	Incorrect policy enforcement	Examples
Context misunderstood	Scripting languages	Mechanism too weak	Privacy in smart cards
Overly restrictive	Changing passwords too frequently	Mechanism is buggy	Buffer overflows in kernels
Noncomprehensive	Executable content in e-mail	Mechanism is misconfigured	Sendmail debug mode

Java type system to enforce the policy that high-security data should never be leaked.[10] Current research in type systems is aimed at eliminating more runtime checks (such as array bounds checks[11]) or type-checking machine code.[12]

Proof-Carrying Code

Proof-carrying code (PCC), a concept introduced by George Necula and Peter Lee,[13] is a promising approach for gaining high assurance of security in systems. The basic idea is to require any untrusted code to come equipped with an explicit, machine-checkable *proof* that the code respects a given security policy. Before executing the code, we simply verify that the proof is valid with respect to both the code and the policy. Because proof checkers can be quite simple (Necula's is about six pages of C code), it is easier to ensure that they are correct. And in principle, PCC can enforce any security policy—not just type safety—as long as the code producer can construct a proof. Necula and Lee have shown that such proofs can be constructed automatically for standard type-safety policies, if a compiler for a type-safe programming language generates the code. Unfortunately, going beyond standard notions of type safety cannot be performed automatically without either restricting the code or requiring human intervention. It is unlikely that programmers will construct explicit proofs. Thus an active area of research is how to integrate compilers and modern theorem provers to produce PCC.

Policy as Achilles' Heel

Thus far, we have focused on technology solutions to the malicious code problem. To be sure, technology can be of service; but there is another critical aspect of the problem that remains to be addressed—the problem of policy.

In current forms, extensible systems do little to determine how a system will behave when extended in certain ways or, put another way, what a particular piece of code can and cannot do. In fact, today's computers are hyper-malleable and overly complicated. This greatly increases the malicious code risk. In the end, determining whether something malicious is happening requires first defining some policy to enforce.

When Policy Breaks Down

Clearly, the notion of policy is deeply intertwined with the concept of malicious code. Understood in terms of policy, the root causes of malicious code fall into two basic categories: bad policy and incorrectly enforced policy.

Bad policy allows malicious code to do something malicious because policy does not forbid it. Even if policy is perfectly enforced by technology, the policy itself must be well formed. Subcategories of bad policy include:

- misunderstandings of context, whereby policy makes no sense in the context where it was applied;
- over restriction, whereby the policy prevents useful work when it is enforced; or
- noncomprehensiveness, whereby policy fails to cover some situation or exists at the wrong level of abstraction.

Incorrect policy enforcement allows code to do something malicious even if it is correctly forbidden by policy. This situation arises when either

- the enforcement mechanism is too weak to implement the desired policy;
- there are bugs in the implementation of the enforcement mechanism; or
- the enforcement mechanism is misconfigured.

Table 2 provides examples of malicious code understood in our policy-based framework.

As an example of context misunderstanding, consider the role of scripting languages such as Visual Basic. Such languages

About the Authors

Gary McGraw is the vice president of Corporate Technology at Cigital (formerly known as Reliable Software Technologies), where he pursues research in software security while leading the Software Security Group. He has served as principal investigator on grants from AFRL, DARPA, NSF, and NIST's Advanced Technology Program. He chairs the National Infosec Research Council's Malicious Code Infosec Science and Technology Study Group. He coauthored both *Java Security* (Wiley, 1996) and *Securing Java* (Wiley, 1999), and is currently writing a book entitled *Building Secure Software* (Addison-Wesley, 2001). Contact him at Cigital, 21351 Ridgetop Circle, Ste. 400, Dulles, VA 20166; gem@rstcorp.com.

Greg Morrisett is an assistant professor in the Computer Science Department at Cornell University. He is a Sloan Research Fellow, recipient of an NSF Career award, member of the IFIP Working Group 2.8 (Functional Programming), editor for the *Journal of Functional Programming*, and an associate editor for *ACM Transactions on Programming Languages and Systems*. His research interests include programming language-based security and type systems. Contact him at the Dept. of Computer Science, 4133 Upson Hall, Cornell Univ., Ithaca, NY 15213; jgm@cs.cornell.edu.

can be can be extremely useful and perfectly safe in some contexts. But in other contexts, scripts can be extremely dangerous. For instance, there is rarely a need for scripts or macros to be run when displaying a document, yet this functionality is exactly what the Melissa virus exploited.

An example of a policy that is too restrictive is one in which users are required to pick new passwords at small intervals. Under such a policy, people often forget their current password. To avoid this, they may write down their password in an insecure place, making it easier for an attacker to steal it.

Most security policies fail to be comprehensive, simply because designers cannot think of all possible attacks. For instance, in the early days of the Internet, there was no need for an e-mail security policy, because mail readers did not interpret messages. Today, messages can contain attachments or scripts that are automatically executed by readers.

Many desirable security policies just aren't achievable or practical. For instance, stringent policies have been formulated for smart cards to prevent disclosure of private or secret information, such as health records or crypto keys. But hiding information is a tricky business and just about any enforcement mechanism will fail to block all information flow off the card. In the case of smart cards, the designers used clever algorithms and packaging techniques to prevent tampering with the card in order to learn private information. But they failed to take into account of the power fluctuations across the connection pins—data that can be used to reconstruct private information.[14]

Sometimes an enforcement mechanism is powerful enough to implement the policy, but its implementation has bugs or weaknesses that prevent it from doing so. The classic example of such bugs are the buffer overflow attacks that arise in operating systems and applications.

Finally, sometimes the enforcement mechanism is powerful enough and coded properly, but simply misconfigured. For example, the sendmail program has debugging features that allow a programmer to gain remote access to a machine. During development, this feature was turned on. Unfortunately, the feature remained on when sendmail was deployed and subsequent attacks such as the Morris worm took advantage of the opening.

Addressing the malicious code problem requires the creation of sound policy and its careful enforcement through technology.

The Many Levels of Policy

System administrators and MIS security people think about policy in terms of user groups, firewall rules, and computer use. Security researchers steeped in programming languages think about policy in terms of memory safety and liveness properties. Government policy wonks think about policy in terms of rules and regulations imposed on users and systems. The problem is, all of these ways of thinking about policy are equally valid!

So how are we to set policy to combat malicious code? We believe the key is to focus on defining metalevel policies that system administrators work with naturally in terms of collections of lower-level enforcement mechanisms. This is no trivial undertaking.

Most of the technologies we've explored earlier in this article can serve to enforce particular aspects of software behavior. Some language researchers, for example, consider the issue of enforcing safety properties "solved," at least in theory. Enforcing liveness properties or confidentiality is harder, but fairly clear research agendas exist to address the open issues. Of course, the terms *safety*, *liveness*, and *confidentiality* have technical meanings. Intuitively, a safety property states that a program will never perform a bad action, for some precisely defined notion of "bad." An example of a bad action is overflowing a buffer. A liveness property, on the other hand, states that a program will eventually perform some desired action or set of actions. For example, the property that a program will eventually release all of the memory that it allocates is a liveness property. Finally, confidentiality is meant to ensure that certain values remain private or secret.

The problem is that low-level properties such as safety and liveness do not align nicely with what most security administrators think of as policy building blocks. Thus an open question is how to express reasonable security policy that can be directly transformed into technology enforcement solutions.

The answer is to understand policy as a layered set of abstractions. Some preliminary work exists (for example Netscape Navigator's approach to policy sets based on expected code behavior), but much work remains to be done.

Our best hope in combating malicious code is creating sound policy about software behavior and enforcing that policy through the use of technology.

The malicious code problem will continue to grow as the Internet grows. The constantly accelerating trends of interconnectedness, complexity, and extensibility make addressing the problem a more urgent need than ever. As extensible information systems become more ubiquitous, moving into everyday devices and playing key roles in life-critical systems, the level of the threat moves out of the technical world and into the real world. We *must* work on this problem.

Our best hope in combating malicious code is creating sound policy about software behavior and enforcing that policy through the use of technology. An emphasis on one or the other alone will do little to help. Any answer will require a set of enforcement technologies that can be directly tied to policy set and understood by nontechnical users.

Acknowledgments

Study Group members include Gary McGraw, Cigital, chair; Avi Rubin, AT&T Research; Ed Felten, Princeton; Peter G. Neumann, SRI; Lee Badger, NAI Labs; Greg Morrisett, Cornell; Tim Teitelbaum, Grammatech; Virgil Gligor, University of Maryland; Tom Markham, Secure Computing; Jay Lepreau, University of Utah; Bob Balzer, ISI; Joshua Haines, Lincoln Labs; Roger Thompson, ICSA.net; Bob Clemons, NSA; Penny Chase, MITRE; Carl Landwehr, Mitretek; Brad Arkin, Reliable Software Technologies; Sami Saydjari, DARPA; Brian Witten, DARPA; and Dave Thompson, Mitretek. Guests who participated in the two day San Antonio workshop include mudge, the l0pht; Crispin Cowen, Wirex; Fred Schneider, Cornell; Peter Lee, CMU; Richard Smith, pharlap; John Rushby, SRI; Dan Wallach, Rice University; Amy Felty, University of Ottawa; and David Evans, University of Virginia.

The workshops on which this report is based were convened under the auspices of the Infosec Research Council (IRC), with members from US Government organizations that sponsor and conduct information security research. Views expressed in the report are those of the authors and may not reflect those of the IRC, its members, or the organizations they represent.

References

1. J. Viega et al., "ITS4: A Static Vulnerability Scanner for C and C++ Code," To appear in *Proc. Ann. Computer Security Applications Conf. 2000*, IEEE Computer Soc. Press, Los Alamitos, Calif: the ITS4 tool is available at www.rstcorp.com/its4.
2. D. Wagner et al., "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Network and Distributed Systems Security Symposium (NDSS 2000)*, Internet Soc., Reston, Va., 2000, pp. 3–18.
3. G. McGraw and E. Felten, *Securing Java: Getting Down to Business with Mobile Code*, John Wiley & Sons, New York, 1999; complete Web edition at www.securingjava.com.
4. J.H. Salzter and M.D. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, IEEE Press, Piscataway, N.J., Vol. 9, No. 63, 1975, pp. 1278–1308.
5. R. Wahbe et al., "Efficient Software-Based Fault Isolation," *Proc. 14th ACM Symp. Operating System Principles (SOSP)*, ACM Press, New York, 1993, pp. 203–216.
6. F. Schneider, "Enforceable Security Policies," *ACM Trans. Information and System Security*, Vol. 2, No. 4, Mar. 2000.
7. U. Erlingsson and F.B. Schneider, "IRM Enforcement of Java Stack Inspection," *IEEE Symp. Security and Privacy*, IEEE Press, Piscataway, N.J., 2000.
8. D. Evans and A. Twyman, "Policy-Directed Code Safety," *Proc. IEEE Symp. Security an Privacy*, IEEE Press, Piscataway, N.J., 1999; see also www.cs.virginia.edu/~evans.
9. U. Erlingsson, U. and F.B. Schneider, "SASI Enforcement of Security Policies: A Retrospective," *Proc. New Security Paradigms Workshop*, ACM Press, New York, 1999, pp. 246–255.
10. A.C. Myers, "JFlow: Practical Mostly Static Information Flow Control," *Proc. 26th ACM Symp. Principles of Programming Languages (POPL)*, ACM Press, New York, 1999, pp. 228–241.
11. H. Xi and F. Pfenning, "Dependent Types in Practical Programming," *Proc. 26th ACM Symp. Principles of Programming Languages (POPL)*, ACM Press, New York, 1999, pp. 214–227.
12. G. Morrisett et al., "From System-F to Typed Assembly Language," *ACM Trans. Programming Languages and Systems*, Vol., 21, No. 3, May 1999, pp. 528–569; www.cs.cornell.edu/talc.
13. G.C. Necula, "Proof-Carrying Code," *Proc. 24th ACM Symp. Principles of Programming Languages (POPL)*, ACM Press, New York, 1997, pp. 106–119; www-nt.cs.Berkeley.edu/home/necula/public_html/pcc.html.
14. P. Kocher, J. Jaffee, and B. Jun, "Differential Power Analysis: Leaking Secrets,"*Advances in Cryptology–CRYPTO'99*. . In M. Weiner, ed., Lecture Notes in Computer Science, Vol. 1666, Springer, New York, Aug. 1999, pp. 388-397; www.cryptography.com/dpa/Dpa.pdf.