

Future Defenses: Technologies to Stop the Unknown Attack

Nicholas Weaver 2002-02-21

Future Defenses: Technologies to Stop the Unknown Attack

by *Nicholas Weaver*

last updated February 21, 2002

Current anti-virus software, combined with sensible filtering (such as quarantining all executable content from e-mail and Web traffic), firewalling, and religiously maintaining patches, serves as a reasonably good defense against the current classes of virus, worms, and script kiddies. This malware includes conventional file infectors, mail worms, people running attack scripts, and active worms based on old security holes. Unfortunately these techniques are not sufficient to stop a [speed-optimized active worm](#) based on a previously undiscovered security hole.

In order to prevent highly damaging "superworms" or hackers using unknown or unpatched exploits, different solutions are needed that are designed to prevent and respond to unknown attacks, rather than known attacks. These techniques need to be automated (as human responses are too slow to prevent damage), implementable, widely deployable, and must require no application changes. This article examines three technologies that offer significant levels of protection against unknown attacks: software fault isolation, intrusion detection through program analysis, and fine-grained mandated access controls.

These technologies share an important property: they do not rely on the correct operation of the programs; rather, they provide a secondary layer of defense should a program be breached and corrupted. It is possible that these systems may also contain flaws; but in order for a successful exploit to occur, both the application and the secondary defense need to be subverted at the same time. Since bugs will continue to be patched, it is much less likely that two overlapping bugs will exist and be known simultaneously than that a single flaw will be known.

Software Fault Isolation

The first technology, Software Fault Isolation (SFI), developed by Wahbe *et al.* ([Efficient Software Based Fault Isolation](#), Proceedings of the Symposium on Operating System Principles, 1993), is a mechanism to create Java-like sandboxes for dynamically-loading arbitrary code in a

language-neutral manner. Unlike JVM-based systems, it can be applied regardless of source language and compiler. The only semantic restriction is that dynamic code generation is not allowed within a fault-isolated module.

SFI operates at the assembly level on a code module through a combination of static analysis and dynamic checks to create the sandbox. The system gives each module its own private memory space in which it is isolated as part of the larger program. The static checks ensure that all statically determinable jumps only occur within the module and to allowed external functions, creating the basic structure of the sandbox.

The dynamic checks are inserted onto every load, store, and dynamic jump (except for those that can be removed through static analysis) to ensure that the program can't escape the sandbox created by the static analysis and allocated private memory. These checks do not ensure fine-grained correctness (such as bounds checking) but only a coarse-grained correctness: the module is not allowed to read or write outside its allowed memory ranges or execute code not contained within the module's code base or allowed external functions. Since the module is now restricted both statically and dynamically, it is effectively sandboxed from other modules within a larger application.

The performance impact, as measured in several scenarios, is relatively minor: the original paper reported a ten percent increase in overall execution time. Later versions reduced this overhead to a little over five percent when compared with unmodified binaries. When using compiled C or C++, this represents a significantly greater level of performance compared to most Java systems while maintaining the same level of protection. Additionally, since it does not require a programmer to recode the application, this technique can be easily adopted into a wide range of systems.

This technique offers two significant advantages: it allows the safe execution of arbitrary binary modules (such as those contained in Active-X and browser plug-ins, which remain a serious potential hole, as Microsoft and others have focused on authentication rather than isolation), and prevents the most damaging effects of buffer overflows: the ability to execute arbitrary code. If a fault-isolated module contains a buffer overflow, the attacker is limited to overwriting data structures and causing the program to jump to an allowed code location. Since the attacker can no longer inject code which the application will run, this eliminates most of the power contained in buffer overflow attacks.

As an example, if IIS used this technique on all its sub-modules and scripts, this would have stopped Code Red. Without the ability to inject arbitrary code into a malformed module (the index server extension in this case), Code Red would have been unable to spread. More importantly, it would prevent a stolen Microsoft private key from being used to create an Active-X based worm because it allows downloaded Active-X modules to be safely sandboxed.

Software Fault Isolation was being commercialized by [Colusa Software](#) between 1994 and 1996 as part of their OmniWare mobile code framework, a means of portably executing binaries in a safe, language and architecture neutral manner. In March of 1996, Colusa was purchased by Microsoft as a means of preventing another Java-like portable code system from entering widespread acceptance. Since then, there have been rumors about continued development within Microsoft but it has yet to be integrated into a released product.

A simplified version, which only checks that jumps are within the allowed ranges, would prevent buffer overflows from executing arbitrary code at an even lower run-time cost. Since it offers a significant improvement in security (roughly half of all major security holes are buffer overflows) at a relatively low performance cost, without requiring language or program changes, this technique should be included at the compiler stage, or contained within a static translator installed on the machine.

Intrusion Detection by Program Analysis

The second major technique, host-based intrusion detection by program analysis, was first proposed and tested by Wagner and Dean ([Intrusion Detection via Static Analysis](#), 2001 IEEE Symposium on Security and Privacy). This IDS performs a static analysis of the program to create an abstract, non-deterministic automata model of the function and system calls. While the program is executing, it compares the system call pattern with a running copy of the automata. If the program ever attempts a system call which violates the model, the system assumes that an intruder has corrupted the program.

Unlike other intrusion detection techniques, which rely on either sample inputs or rule sets, this method has a provable zero false positive rate, eliminating all false alarms. This means the intrusion detection system can initiate automatic responses: blocking the system call, shutting down the corrupted program, and alerting the administrator. The zero false positive rate is due to the programmatic nature of the IDS, which contains a model that represents all possible legal paths through the program, ensuring that any detected deviation from the model is not

caused by the program's code but by code inserted by a bug or an attacker.

The biggest concern is the very high run time required for the IDS to operate. This is due to imprecision in the model of execution because the IDS only has access to system call information from the running program. There are two solutions: the first is to have the IDS walk the stack to determine the program's call state before allowing a system call, which greatly increases the precision. The second, which feeds even more information, has the IDS that is modifying the binaries to various function calls send information about their invocation to the IDS system.

The other major limitation is the inability to handle multi-threaded programs without an explicit mechanism to detect the occurrence of a thread switch. This is not a problem for most UNIX programs, but is a significant problem if one wishes to apply this technique to Windows systems, which rely more heavily on user threads. Yet the same performance-enhancing solution of program annotation can be used to overcome this limitation by transmitting when thread switches occur.

A Code Red-style worm would be able to break into a system protected by such an IDS, but the potential damage would be greatly limited. It would be easy to deface Web pages by substituting the routines that deliver the content to pages that return erroneous content although even though they may behave identically at the system call level. Since such behavior is not contained in the original program, the IDS would stop the program before damage could be done.

The Security Enhanced Linux (SELinux) Project

The final technique utilizes the fine-grained mandatory access controls that are the primary foundation of the [Security Enhanced Linux](#) (SELinux) project, a research design created by the National Security Administration's Information Assurance Research Group. SELinux is designed around two key concepts: fine-grained mandatory access controls and a separation between enforcement mechanisms and policies.

Most systems have very crude levels of access controls: a program running as root (or the OS equivalent) has access to every function in the machine and has nearly unlimited ability to manipulate the computer's systems. Programs run by a user usually inherit the user's ability to manipulate files. To make matters worse, necessary functions often require root level

privileges: as an example, sendmail in Unix systems needs to be able to bind to port 25 and write to the user's mailspools while the other root abilities (such as being able to put Ethernet ports into promiscuous mode) are not. Yet a corruption of sendmail (from a worm or attack script) has access to abilities that sendmail never needs to use. This problem is most clearly demonstrated by Code Red II: because IIS runs at system level, as are processes it can execute, the backdoor installed by Code Red II can perform practically any action desired, including shutting down the infected machine or even erasing the disk.

Fine-grained mandatory access controls solve this problem. Instead of privileges being a crude measure of access, fine-grained controls ensure that every system call is authorized as a function of the running program, the user running the program, and the context that invoked the program. This can be a very powerful tool if the enforcement policies are well written. One policy could have a Web server that could only read (but not write) to the filesystem as an anonymous user, with the ability to write to only specific log files. With similar restrictions on network and other activities, the scope of a Code Red-style worm would be severely limited: it could infect a Web server, but it couldn't initiate new connections in order to spread itself or execute any program except those which the Web server normally is able to run.

Additional access controls could specify whether the stack and heap are allowed to contain executable code. Modern processors can enforce this separation through the virtual memory system, but only if the OS requests it. Except for run-time compilers and dynamic code specialization, most programs have no need to generate dynamic code. Thus for the majority of the programs, the OS could simply ensure that heap and stack memory is non-executable, eliminating most of the power of buffer overflow attacks with no effect on performance.

This technique requires small but numerous changes within the operating system. Almost every system call needs to enforce security policies, but coding the policies into the checked calls results in an inflexible system. Instead, SELinux uses a "security server", a separate area of the operating system that performs the checks. Whenever a portion of the operating system needs to check if a command is allowed, it calls the security server to verify that the current policy allows for the action to proceed. This allows for maximum flexibility as different policies can be given to the security server and global security policies can even be changed while the system continues to run.

The greatest advantage of this technique is that it offers significant flexibility, a high level of protection, and high efficiency. The flexibility comes from the ability to write arbitrary policies

for programs or users with different goals, from providing basic isolation for some servers to minimizing the risks of viruses by limiting the power of unknown executables. The high level of protection is derived from the ability to compose comprehensive security policies that directly limit an application to the tasks it needs to complete. Finally, the efficiency is derived from the observation that, for most programs, the number of system calls is actually quite low. So, although the checks incur a significant (up to twenty-five percent) penalty for system call intensive synthetic benchmarks, real-world benchmarks such as Web serving or kernel compilation have an under five percent impact in total execution time. If a system set-up was not concerned with security (a dubious proposition) this overhead could be almost entirely eliminated by having a dummy security server.

The two major disadvantages are that policies need to be carefully composed and that the mechanisms need to be built into the operating system. Well-constructed default policies make an excellent starting point, but highly secure systems will require a fair degree of customization to achieve the best results. The OS integration itself is not difficult, but must be done by those who have access to the source code. If Microsoft is truly serious about improving security, fine-grained mandatory access controls could provide a powerful, flexible, and efficient technique.

Conclusion

All three techniques discussed in this article are of significant importance - one or more of them should be integrated into future systems. Although in the Windows world only Microsoft could add in fine-grained mandatory access controls, the other methods could easily be applied by third parties. Since both program analysis-based intrusion detection and software fault isolation can be done by manipulating compiled programs, these offer language-neutral and application-neutral solutions without changes to either the programs or the operating system. Fine-grained access controls require OS modification but do not require changes to the applications.

Security products that use one or more of these techniques should be highly attractive to those wishing to install secure network systems. Although none apart from SE Linux are currently available, they offer significant potential for greatly improved security in the near future and some are probably in early development.

Nicholas Weaver is a computer science graduate student at the University of California, Berkeley. His primary interest is FPGA and computer architecture, with secondary interests in computer security, molecular computation, and cryptographic implementations.

[Privacy Statement](#)

Copyright 2006, SecurityFocus