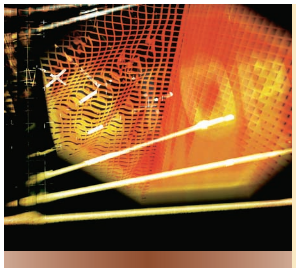
"Patch on Demand" Saves Even More Time?

Angelos D. Keromytis, Columbia University

In the June 2004 Security column ("A Patch in Nine Saves Time?"pp. 82-83), Bill Arbaugh makes two interesting observations: first, whoever has the tightest observe-orient-decide-act (OODA) loop will prevail in a confrontation; second, the infection rates of recent worms suggest that the good guys are losing the battle.



A vaccination system could automatically generate patches to protect an application's source code.

Arbaugh offers some sensible suggestions to vendors and security professionals on improving patch management. However, the best indication that we are losing the battle is not the infection rates of worms such as Slammer and Blaster, but the shrinking interval between discovering and announcing a new vulnerability and the appearance of a worm or attack that exploits it.

The most recent example is the Witty worm, which effectively exploited a vulnerability present in a small population of hosts—approximately 12,000 computers. Although it was first discovered on 8 March 2004, the vendor didn't announce the vulnerability until 18 March, after it had made a patch available. A little over a day after the announcement, Witty made its first appearance.

ZERO-DAY ATTACKS

Given such a short turnaround time, we can reasonably expect to soon experience a *zero-day* worm. Zero-day attacks are those for which users receive no prior warning and thus have no preventive measures in place.

To date, a combination of aggressive packet filtering and proactive application patching could—at least in principle—defeat all the worms we have encountered. Although we could, in theory, deploy patches and network filters automatically, the practicality of employing such measures and their effect on regular system operation are an entirely different story.

Witty came close to being a zero-day worm; for most organizations, it was. Few system administrators had even seen the announcement before the attack, much less downloaded and installed the necessary software patch.

Furthermore, as Arbaugh ("Windows of Vulnerability: A Case Study of Analysis," *Computer*, Dec. 2000, pp. 52-59) and others such as Eric Rescorla ("Security Holes … Who Cares?" *Proc. 12th Usenix Security Symp.*, Usenix, 2003, pp. 75-90) have noted, many administrators find it impractical, if not otherwise unacceptable, to patch or upgrade their systems when a vulnerability is announced. Instead, they wait for news of an actual exploitation. In many cases, this is simply too late.

PATCH ON DEMAND

What to do then? One new idea is to integrate the vulnerability discovery, patch generation, and patch application cycles into a system that would automatically detect a new attack, analyze its modus operandi, determine the best software patch, and apply it at the desired level of granularity—LAN, enterprise, or Internet-wide.

Although the system would still function by reacting to attacks, its response time would be significantly shorter. Perhaps most importantly, it could operate autonomously.

Furthermore, by retaining a focus on software patching, this approach could avoid at least some pitfalls of network-based defense techniques. These techniques, which include packet and content filtering, can fall prey to worms that exploit opportunistic encryption, polymorphism, or metamorphism. Traditional viruses use polymorphism to make detection more difficult: A small decoder, which changes periodically, decrypts the virus's main body prior to execution.

Metamorphic viruses, on the other hand, completely translate the virus code to use different but equivalent instructions every time it infects a new target. Classic signature-matching techniques are, for the most part, incapable of detecting such viruses.

Metamorphism, in particular, makes detecting new viruses extremely difficult and time-consuming. Worse, several advanced metamorphic virus engines, such as Zmist, have been publicly available for a few years. I suspect that their use in a future worm is only a matter of time. One reason we have not seen them yet may be that current practices are so effective in creating highly infectious worms.

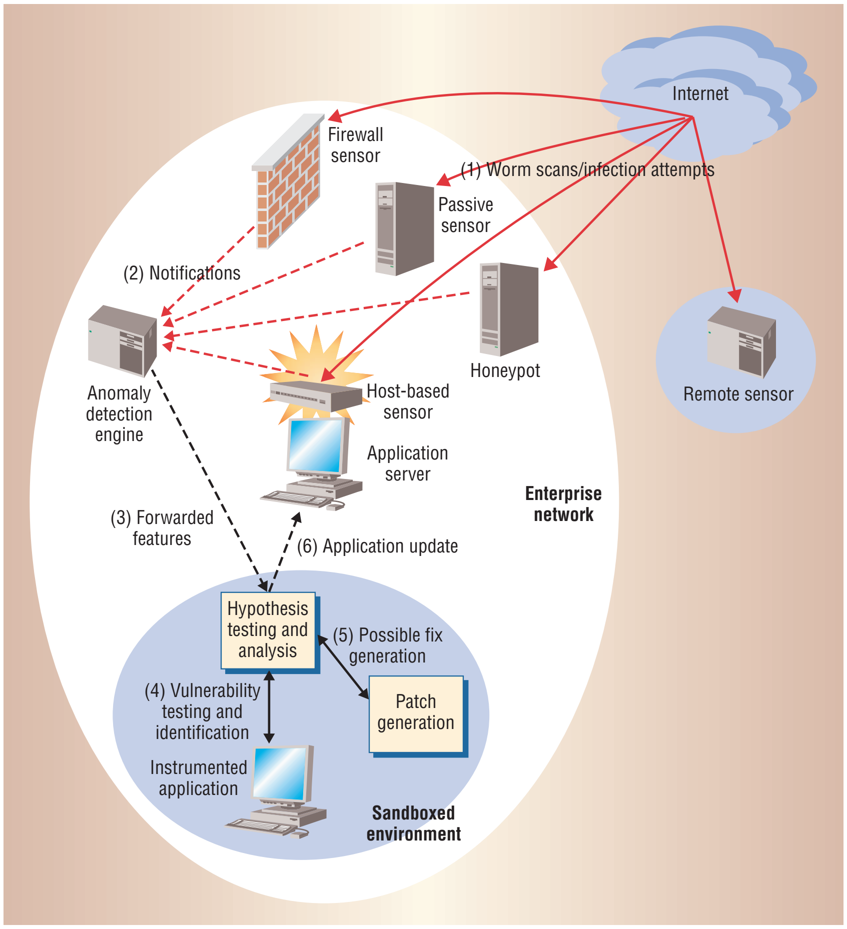


Figure 1. Worm vaccination architecture. (1) Sensors deployed at various locations in the network detect a potential worm and (2) notify an analysis engine. (3) The engine forwards the potential infection vector and other relevant information to an isolated environment, where (4) the potential infection vector is tested against an appropriately instrumented version of the targeted application and the vulnerability is identified. (5) Several software patches are generated and tested using different heuristics. If one of the patches is found to be immune to the infection, (6) the application server is updated.

Vaccination system

Can we develop a patch-on-demand system that is both practical and safe? This is the question we are investigating at Columbia University's Network Security Lab (http://nsl.cs.columbia.edu/). We believe that some of the necessary technology, at least for the initial steps, is already available.

To begin with, we think it is much easier to identify a previously unknown vulnerability by observing the actions of an attacker, such as a worm, that must "demonstrate" its use (and side effects) while attacking a new system.

Figure 1 shows an automated worm vaccination system that includes a *honeypot*—a security resource that pretends to be an easy target for purposes of attracting and exposing attackers. Using honeypots for early-warning systems has increased over the past few years. A properly instrumented vaccination system could monitor the attack activity at the software level and detect previously unknown attacks against what the adversary considers a valid target for infection.

As the worm attempts to infect the system, it reveals not only the vulnerability's existence but also its precise characteristics. For example, in the case of buffer overflows, a trace of the stack shows up at the time of the attack, along with identification of the routine where the overflow occurred, the vulnerable buffer, and the attack vector.

Developers could use this system to get the information needed to develop a patch manually. We propose going a step further. By applying a series of transformations to the vulnerable application's source code, we want to automatically create a narrowly defined patch that protects against the specific vulnerability. At its simplest, such a patch would recompile the vulnerable part of the application with a mechanism like StackGuard.

More advanced fixes are also possible. For example, we have been experimenting with patches that maintain the application's availability (http://nsl.cs.columbia.edu/projects/wormv/).

Results

Our preliminary experiments with several vulnerable open source applications show a better than 80 percent success rate in protecting against an attack and maintaining continuous operation. In the remaining cases, the patch successfully thwarts the attack, but the application terminates its execution.

An analysis of an Apache server that our system patched in response to a Slapper-like attack showed no impact on performance. Furthermore, the patching process took less than 10 seconds. Naturally, additional research is needed to determine the mechanism's limits with a richer corpus of vulnerable software.

CHALLENGES

In our exploratory work we have already identified issues that require further research, such as reasoning about the system's reliability and security, extending it to cover other types of software failures and attacks, and handling binary-only systems—to name but a few.

Risk

As with any fully automated task, the risks of relying on automated patching and testing as the only real-time defense techniques are not fully understood. Furthermore, it should be obvious that this system does not address the growing number of e-mail worms that spread by exploiting other types of security flaws and human curiosity.

Software engineering, programming languages, security, and networking are all components in developing the system. At the very least, some of the techniques we have developed should help in the traditional "penetrate and patch" cycle.

Trust

At a different abstraction level, these ideas could serve as existential proof for reactive decentralized mechanisms that do not require collaboration but can respond through purely local action to global phenomena such as worms.

Several researchers have proposed the need for large-scale collaborative mechanisms to detect and filter future worm attacks. For example, Vern Paxson proposed a Cyber Center for Disease Control for identifying outbreaks, rapidly analyzing pathogens, and fighting the infection ("How to Own the Internet in Your Spare Time," *Proc. Usenix Security Symp.*, Usenix, 2002, pp. 149-167). The CCDC could use our mechanism to quickly create needed patches.

The trust issues inherent in such an approach present unique challenges. It seems unlikely that many organizations would trust such a third entity to dynamically alter their own defensive posture—for example, by pushing network filters to network firewalls or sending software patches.

A CCDC whose role was confined to managing a large number of honeypots and attack-detection sensors would be valuable in spreading the word of new attacks. Organizations, or their security service providers, could run software vaccination systems indepen- dently of each other. All that this hypothetical CCDC would do is broadcast the newly captured attack vector and allow individual organizations to develop their own patches in real time: Trust but verify! ■

Angelos D. Keromytis is an assistant professor in the Department of Computer Science and director of the Network Security Lab at Columbia University. Contact him at angelos@cs.columbia.edu.

Editor: William A. Arbaugh, Dept. of Computer Science, University of Maryland at College Park; waa@cs.umd.edu