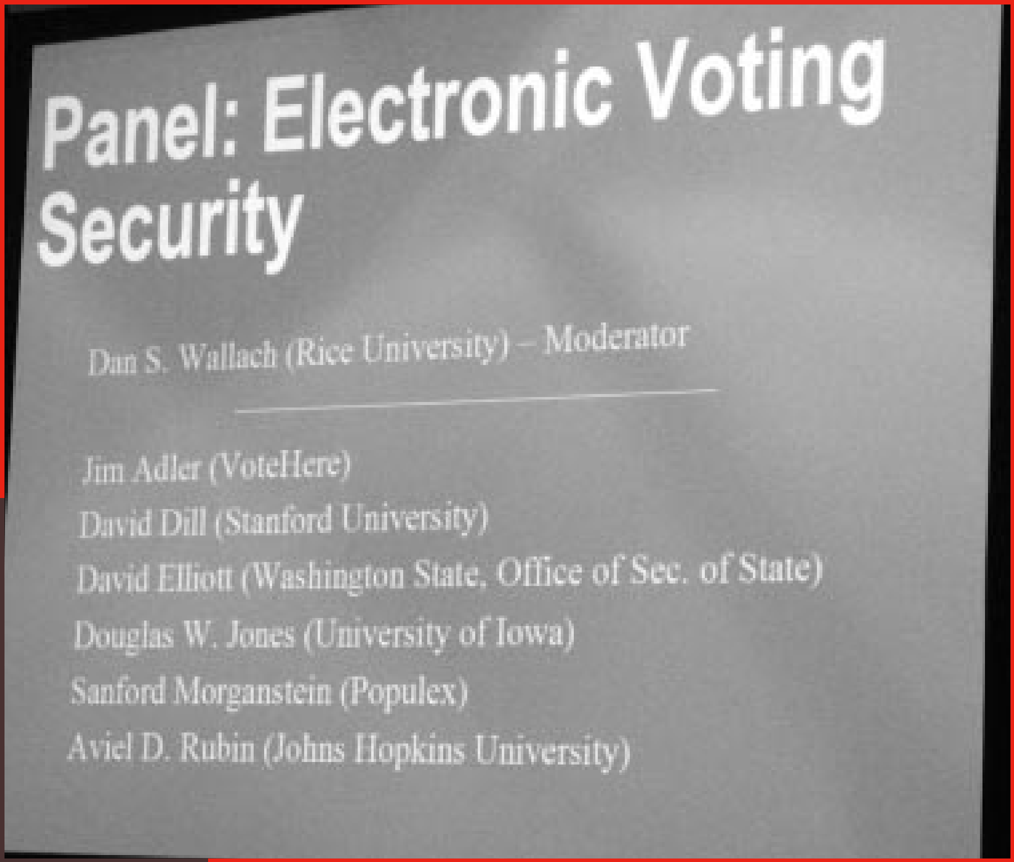
# ;login:

THE USENIX MAGAZINE

December 2003 • volume 28 • number 6



Panel: Electronic Voting Security

Dan S. Wallach (Rice University) – Moderator

Jim Adler (VoteHere)
David Dill (Stanford University)
David Elliott (Washington State, Office of Sec. of State)
Douglas W. Jones (University of Iowa)
Sanford Morganstein (Populex)
Aviel D. Rubin (Johns Hopkins University)

{

inside:

SECURITY

**Perrine:** The End of crypt() Passwords . . . Please?
**Wysopal:** Learning Security QA from the Vulnerability Researchers
**Damron:** Identifiable Fingerprints in Network Applications
**Balas:** Sebek: Covert Glass-Box Host Analysis
**Jacobsson & Menczer:** Untraceable Email Cluster Bombs
**Mudge:** Insider Threat
**Singer:** Life Without Firewalls
**Deraison & Gula:** Nessus
**Forte:** Coordinated Incident Response Procedures
**Russell:** How Are We Going to Patch All These Boxes?
**Kenneally:** Evidence Enhancing Technology

BOOK REVIEWS AND HISTORY

USENIX NEWS

CONFERENCE REPORTS

12th USENIX Security Symposium

## Focus Issue: Security

Guest Editor: Rik Farrow

USENIX

The Advanced Computing Systems Association

# learning security QA from the vulnerability researchers

**by Chris Wysopal**

Chris Wysopal is the vice president of research and development at @stake, where he leads research on how to build and test software for security vulnerabilities.

*cwysopal@stake.com*

Every day, vulnerability researchers find and publicly disclose new vulnerabilities for software products. Many of these products are made by vendors who assure us that they know how to create secure software. What makes it possible for a vulnerability researcher, who usually doesn't have access to design documentation or source code, to find these problems? He would seem to be at a major disadvantage compared to the vendor's testing team. Is the vendor not looking? Or is there something about the process of discovering vulnerabilities that keeps software vendors from doing it well? This article will take a look at the differences between researcher and vendor and how these differences lead to different results.

First, let's examine a bit of history. The setting is the 1997 USENIX security conference. Mudge from the L0pht is hanging out with Hobbit, who, while not officially part of the group, often collaborated with it. The two are approached by Paul Leach, a Microsoft security architect, and other senior technical people from Microsoft. The gentlemen from Microsoft wanted to sit down and learn how Hobbit had discovered vulnerabilities in the Windows CIFS protocol[1] and how Mudge managed to find flaws in Windows NT's network authentication. Over a fine dinner and a few bottles of wine, the two researchers took the Microsoft security folks through the process of black box reverse engineering, with a vulnerability twist. This is what they described.

1. Hobbit, "CIFS: Common Insecurities Fail Scrutiny," 1997, *http://downloads.securityfocus.com/library/cifs.txt.*

The first task for attacking the CIFS protocol was to install a host with a sniffer on the network between two hosts running Windows. The sniffer was running on a non-Microsoft OS. The reason for this wasn't just because Hobbit's coding skills happened to be better on UNIX. It was also to make sure that the analysis host's OS wasn't contaminated with any Windows internals knowledge. If a Windows host was used for analyzing the network traffic, the network stack or other internal OS components might perform hidden data manipulation. This is a theme that will pervade the reverse engineering process. All analysis tools, whether off-the-shelf or custom coded, need to be free of unwanted interaction with the program under test. For network analysis, a different OS often provides enough isolation. For analysis programs that must run on the same host, it is best to avoid using higher-level OS APIs that might perform unwanted data manipulation.

With the sniffer in place, CIFS transactions were performed between the two Windows hosts, and the network packets were recorded. Transactions were repeated with slight changes, and the differences in the packets were noted. This was a laborious process, but over time it was clear what different packet types there were and what data fields were in these packets. The gentlemen from Microsoft were surprised by the approach. It had never occurred to them to analyze the protocol this way. After all, they had Windows API functions they could call to look at the data in the CIFS transactions. They had design documentation to tell them what was in a particular field within a packet. Their analysis approach differed in that they were seeing how the CIFS protocol was supposed to work, not how it actually did work. With his independent-analysis approach, Hobbit was able to discover workings of the CIFS protocol that were unknown to its designers and implementers.

It is hard to believe in this age of heightened security awareness that most people who develop software still don't know how vulnerability researchers work.

What may seem like liabilities on the vulnerability researchers' side – using a different OS, having no design documentation or code to look at, and having no access to internal testing tools (which may share code with the system under test) – are turned into benefits. The researcher is not tempted to take a time-saving shortcut while analyzing the system. He must build up from scratch what the bits on the wire mean and how they can be changed. He gets an unbiased view of how the program actually works.

It is understandable that the security folks from Microsoft didn't know how vulnerability researchers worked in 1997. Vulnerability researchers were a small, closed group of people dealing with something fairly arcane. Today, software security affects every computer user, from those in the military and government to the teenager at home. It is hard to believe in this age of heightened security awareness that most people who develop software still don't know how vulnerability researchers work. The software-testing community needs to learn why these researchers are successful and start to work like them, though perhaps a bit more formally. Otherwise, vulnerabilities that could have been found before a product is delivered to the customer will be found by researchers and end up needing to be patched, or worse, be exploited.

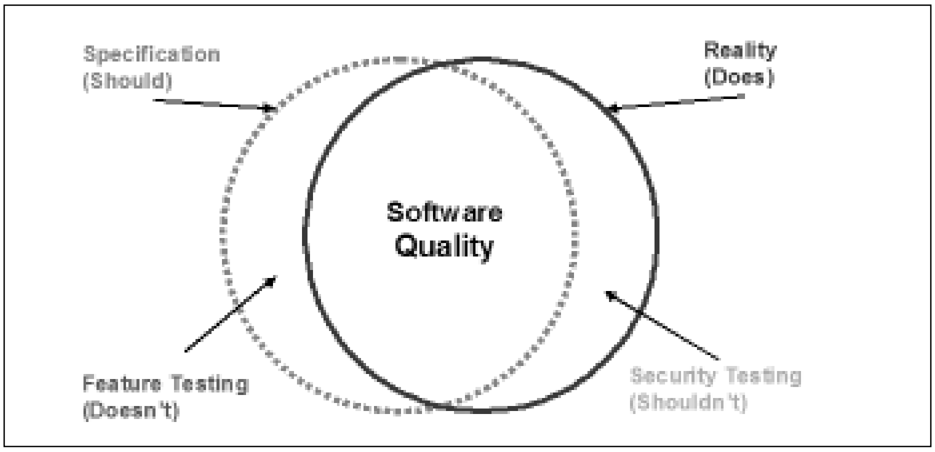
The first thing the researcher does is zero in on the weakest links in the software, the areas of highest risk. Ad hoc threat modeling is performed on the data as it flows in and out of the program. Where can the attacker inject data into the program? Where are the places that data can be injected without first performing authentication? This is the primary attack surface.

Many software testers, when they actually take time to do security testing, get bogged down looking at the security throughout the entire application. This is understandable, since they are used to testing the functionality of the whole application. But security testing is very different from feature testing. When there is limited time, and there always is, there is a need to start at the areas of highest risk and continue toward areas of lower risk.

The other major shift that testers need to make is to stop thinking of security as a feature that can be tested with positive functionality testing. Positive testing is making sure a feature works. If the program has authentication and access control lists, those are typically tested. Vulnerability researchers almost never look at the security features. Positive testing will not find out that a program contains a stack buffer overflow in code that reads data from the network. Testers need to learn the art of negative testing, the art of causing faults.

In the perfect development world, a finished program would exactly match the functionality of its design specification, with no more functionality and no less. In the graphic on the facing page there are two circles, one representing the program's design and the other representing the actual implementation. Since we have not perfected software coding, there is a need for testers to find the mistakes that coders make. These are the mistakes that lead to the design not matching the implementation. But most testers only cover the deviation, where the implementation is lacking functionality defined in the design. What about the deviation where the program has functionality that was unintended? This is where the vulnerability researcher's skills come into play. This is where knowing what a program actually does and not just what it was designed to do is critical.

Negative testing is identifying the inputs of the program and putting in data that is obviously invalid. Typically, the data is nowhere close to what a normal user would do.



*Design vs. Reality*[2]

2. H. Thompson and J.A. Whittaker, "Testing for Software Security," *Dr. Dobb's Journal* (November 2002).

Many testers can't imaging anyone inputting a username of 50,000 characters, so they don't make it part of their test plan. But attackers do try such seemingly ridiculous inputs, so the negative testing plan should too.

Luckily the tester does not need to input this manually. State-of-the-art vulnerability research involves automated fuzzers that can perform the fault-injection process. Fuzzers have a list of rules for creating input that is known to cause errors in processing: long strings, Unicode strings, script interpreter commands and delimiters, printf-style format strings, file names, etc. @stake's WebProxy tool does this for HTTP. Immunity Security's SPIKE is a fuzzer-creation toolkit that can be used to fuzz arbitrary network protocols. Once you have a fuzzer for the protocol you want to test, you just run it against a debug version of the software running in the debugger. Chances are the program will eventually crash and you will be sitting at the vulnerable line of code in the debugger.

There is much more for the software tester to learn, but a great start is to learn how to threat model for the highest-risk attack surfaces, understand negative testing, and get up to speed on fuzzing. Follow the online security community to learn the tools and techniques that vulnerability researchers use and make them part of your quality assurance process. Every vulnerability found during QA is one less for the vulnerability researchers to find and one less vulnerability for software users to patch.