Inside the Windows Security Push

The Microsoft Windows development team spent two months in 2002 analyzing product design, code, and documentation to fix security issues. The results of this security push include a new process and several lessons learned for future projects.

MICHAEL HOWARD AND STEVE LIPNER
Microsoft

During February and March 2002, all feature development on Windows products at Microsoft stopped so that the complete Windows development team could analyze the product design, code, test plans, and documentation for security issues. Our team at Microsoft named the process the Windows Security Push. The security push process was derived from the .NET Framework Security Push project conducted in December 2001.

Most Windows feature teams finished the Windows Security Push in March, and since then, the company has performed many other pushes across its product line including SQL Server, Office, Exchange, and others. This article outlines the Windows Security Push process in detail as well as the rationale behind it and some of the lessons learned from it.

The rationale for the push

The driving force behind the Windows Security Push was Bill Gates' "Trustworthy Computing" memo of January 2002 (see www.microsoft.com/presspass/exec/craig/10-02trustworthywp.asp for a copy). In that memo, Gates outlined the need for more secure platforms on which to build future solutions. These platforms should not only be secure from malicious attack, he explained, they should be more robust and afford users the privacy they demand.

Another reason for the Windows push was the highly successful .NET Framework push, which predated the Gates memo. That push's intent was to uncover any remaining security issues before final release. The team behind it felt that one big push at the end (in which all team members spent time focusing on nothing but security) would be worthwhile, even if it meant slipping the schedule.

And it was worthwhile: the team made a few important changes to the .NET Framework including placing the State Service (an optional component used to maintain HTTP state) and the ASP.NET worker process (used to create dynamic content on a Web server) in lower privileged accounts, rather than SYSTEM. Running code as SYSTEM gives administrators minimum hassle, but such a configuration ignores the principle of "least privilege." If hackers were to compromise the server, for example, they could have used the ASP.NET process to run attack code with SYSTEM privileges. Thus, changing the default configuration so that ASP.NET runs with minimal privileges was a good idea.

The team also let administrators opt in support for downloading and executing Internet-based code rather than executing such partially trusted code with no interaction. Many threats come from malicious Internet-borne code, so it made perfect sense to provide that functionality as an option, not as a default.

A whole new generation of threats exists on the Internet, which is another critical reason for the Windows Security Push. Technology that was secure eight years ago is not secure enough today. A good example is NetDDE, which is a mechanism for exchanging data between applications over a local network. NetDDE was appropriate when it was conceived in the days of small workgroups in which all users on a LAN could be presumed to be trusted. The protocol, however, does not support authen-

Microsoft constantly hires new employees, and frankly, most of them have little knowledge of how to build secure software.

tication, privacy, or tamper detection. Because PCs are more interconnected now, we disabled the NetDDE service by default in the Windows .NET Server 2003.

Although the Windows Security Push was modeled on the .NET push, there was a big difference: scale. Roughly 8,500 people comprise the Windows team—about 1,000 comprise the .NET Framework team. In later sections, we discuss how we scaled the process to deploy the entire Windows team in a coordinated securing effort. But first, let's look at what the security push process involves. At a high level, it consists of four distinct needs:

- Education
- Analysis and review
- Constructing threat models
- Making design changes

Education

The first part of any security push must be education, because we cannot assume that everyone understands how to build secure software. In our experience, many developers think "secure software" means "security features." This assumption is simply wrong. Secure software means paying attention to detail, understanding threats, building extra defensive layers, reducing attack surface, and using secure defaults. Layering an application with security technology won't fix implementation issues such as buffer overrun vulnerabilities. Furthermore, such technology will not render a design secure if it does not mitigate the primary threats to the system.

Although the Secure Windows Initiative team provided security training to component development teams across the Windows division during the year before the security push, we believed that ensuring a common level of understanding via division-wide mandatory training would be worthwhile.

We provided three training tracks during the push: one for designers, program managers, and architects, another for developers, and a third for testers. Documentation people attended the track appropriate for the documentation they were supposed write. Each training session occurred five or six times, with a total of 8,500 people attending over a 10-day period. Approximately 12 percent came to more than one training session. We broke the security push training into two halves: the first half outlined security push logistics, organization, and process, and the second covered technical material appropriate to the class attendees' role.

Designers

The training reinforced the mindset of doing whatever it takes to reduce the product's attack surface. Our view is that if a feature is not enabled by default, it is much less likely to be attacked; if that feature runs with reduced privilege, then the potential for damage reduces even more. We saw the effect of such thinking in the Windows .NET Server 2003 project—the number of features enabled by default is greatly reduced compared to Windows 2000. We know that we didn't reach perfection and that vulnerabilities will still exist in Windows .NET Server, but we also know that the overall potential damage is reduced because so many features no longer run by default. Reducing this number also pays off by reducing the urgency with which an IT manager needs to apply patches when vulnerabilities appear. A patch is still issued, but the urgency to deploy the fix is reduced. There is a downside, though: features that previously ran by default now must be enabled if the administrator needs access.

Developers

In the training, developers learned the golden rule of never trusting input. There comes a point in the code at which data from an untrusted source is deemed well formed and correct. Until that point, a developer must be very careful with the data. In our experience, most security vulnerabilities are due to developers placing too much trust in data correctness. Common practice is to teach developers to stay away from certain coding techniques and function calls, such as the much-maligned C `strcpy` function when considering buffer overrun issues. The problem is not with `strcpy`, but with the developer trusting the source data to be no larger than the target buffer's size. Simply teaching people to stay away from certain function calls, rather than educating them about why the function is dangerous, leads to only marginally more secure products.

Testers

The principal skill we taught testers was data mutation or *intelligent fuzzing*. Data mutation involves perturbing the environment such that the code that handles the data entering an interface behaves insecurely. The process involves deriving application interfaces from the application's threat model (threat modeling is discussed later in this article), determining the data structures that enter the interfaces, and mutating that data in ways that cause the application to fail. Example perturbing techniques include wrong sign, wrong data type, null, escaped data, out

of bounds, too long, too short, and zero length data.

A good threat model includes an inventory of attack targets in the application, which is an incredibly useful starting point for the tester. For each threat in the threat model, there should be at least two security tests: a "white hat" test to verify that mitigation techniques work and a test that tries to find ways to make them fail. Threat-mitigation methods such as authentication, authorization, and encryption have broad attack techniques associated with them. A tester, for example, could attempt to circumvent authentication by constructing a test tool that attempts to downgrade to a less secure authentication.

We also urged testers to run tests under a nonadministrative account. We did this for two reasons: to uncover usability issues and to test cases in which applications fail because the code was written and tested using administrative accounts. Luckily, this failure mode is relatively rare. However, the second reason is directly useful from a security-testing standpoint: testers should attempt to compromise the system as a nonadministrator. Compromising the system from an administrative account is a moot point: if the tester is already an administrator, he or she already owns the system.

Improved training courses

We realized that one training course was not enough. The purpose of the training was not to teach designers everything about building secure software—rather, it was to raise awareness, identify common mistakes, and teach security basics. The following weeks of the push served as homework or "projects courses." During this time, team members had the chance to exercise what they had learned and had access to "tutors" from the Secure Windows Initiative team.

We also gave all product group members copies of *Writing Secure Code*[1] and directed them to it as their first point of reference. We knew we would get many questions during the course of the push, and the book helped immensely to lighten the load on the Secure Windows Initiative team members.

The educational material was augmented with best-practice checklists for developers, testers, and designers. Although these checklists are useful, we feel they are a minimum security bar.

We also realized that our training of current employees was not enough. Microsoft constantly hires new employees, and frankly, most of them have little knowledge of how to build secure software. Thus, we now have security bootcamp training for new product group employees.

Analysis and review

The first part of the push process proper was to build an inventory of source code files and assign a developer to review each one. During the security push, developers must review the code, looking for security flaws and file bugs. The most productive way to review code is to follow the data flow through the application, questioning assumptions about how the data is used and transformed. However, the going is slow—about 3,000 lines per day to perform the task effectively.

We also used tools developed by Microsoft Research to analyze code. These tools help detect common code-level flaws such as buffer overruns and are constantly updated as we learn about new vulnerability classes. Although these tools are useful and improve the process's scale and efficiency, they cannot replace a well-educated brain taught to find security bugs.

We made an interesting discovery during the security pushes, domain experts—when instilled with a degree of security expertise—found new classes of issues as they looked at their bailiwick in a new light. We document many of these findings at http://msdn.microsoft.com/columns/secure.asp.

Constructing threat models

The prime deliverable for designers was threat modeling. We believe that understanding threats is the only way developers can build secure software. Once they understand the threats to the system, they can determine if the threats are appropriately mitigated. Threat modeling follows a set, but not new, procedure:[2]

1. *Decompose the application*. The first phase is to determine the system's boundaries or scope and understand the boundaries between trusted and untrusted components. UML activity diagrams and dataflow diagrams are useful for such decomposition.
2. Determine threat targets and categories. The next step is to take components from the decomposition process, use them as the threat targets, and determine the broad threat categories to each target. The concept we use to define threat types is called STRIDE (spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege). Think of STRIDE as a finer-grained version of CIA (confidentiality, integrity, and availability).
3. *Identify attack mechanisms*. The threat tree (derived from hardware fault trees) describes the decision-making process an attacker would go through to compromise a component. When the decomposition process gives an inventory of application components, we can start identifying threats to each of them. Once we identify a potential threat, we then determine how that threat could manifest itself via threat trees. Each attackable threat target will have one or more threat trees outlining how the component could be compromised.
4. *Respond to the threats*. Finally, we need mitigation techniques for each threat. The threat type drives mitigation techniques—for example, spoofing threats can often be solved by authentication, tampering threats by encryp-

We made a number of similar design changes with the aim of improving the ability of Windows to remain secure.

tion, message authentication codes, digital signatures, or access control mechanisms.

We found threat modeling to be of great benefit for many reasons. It

- Helps determine the main threats to the system, the categories of threats, and whether the threats are mitigated
- Finds different issues than code reviews usually find, such as bugs
- Can drive the testing process

As design teams completed their threat models, we used them to focus code review and testing on components in which the threat models showed that errors would most likely result in security problems. These focused review and testing efforts were especially productive in directing our efforts toward key issues, such as enforcing least privilege and disabling unused features.

Making design changes

The threat modeling process was also instrumental in directing the Windows team toward design changes that would eliminate vulnerabilities and reduce Windows' susceptibility to attack. In some cases, we made specific changes that eliminated product vulnerabilities. In others, as discussed earlier, design changes reduced the attack surface by disabling features via default or reducing privilege.

We also introduced design changes that would improve the resistance of Windows to attack even when vulnerabilities remained. For example, we changed Internet Explorer's default behavior so that frame display is disabled in the browser's "restricted sites" zone. Because Microsoft email programs display HTML email messages in the restricted sites zone by default, this change eliminates HTML email as an attack vector to exploit vulnerabilities in the frames mechanism.

Also, designers added a new privilege that restricts which accounts can impersonate other accounts. By default, only administrators and services have this privilege. The threat is small but real: if a less-privileged user could install some malware on a computer and convince a more privileged user to connect to it using an interprocess communication method such as named pipes, the rogue application could potentially impersonate the more privileged account. This is no longer the case in Windows .NET Server 2003.

We also disabled by default or moved to lower privilege accounts more than 20 system services.

We made a number of similar design changes with the aim of improving the ability of Windows to remain secure even in the presence of vulnerabilities undetected during the security push. For example, Windows .NET Server 2003 is now compiled with the Microsoft Visual C++ -GS flag, which helps detect some forms of stack-based buffer overruns at runtime. (For more information, see http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vctchCompilerSecurityChecksInDepth.asp.)

Sample code

Each Windows version includes several software development kit (SDK) code samples, which help customer development staffs build custom applications that can use the product's features. We know that many customers begin with our sample code and modify it as needed to implement their applications. We realized early in the security push process that insecure sample code can lead to insecure end-user applications, so we categorized all sample code as requiring intensive security review to provide customers with guidance on how to build safe and secure applications.

Sign off

The security push process identified an individual "owner" for each source file in Windows. The security push ended when each source file was "signed off" by its owner as having been reviewed to a depth appropriate to the file's exposure (as determined in the threat models). This sign-off process makes the role, importance, and accountability of individual contributors' efforts especially clear.

Targets

All security pushes have followed a common model when determining which code base to target: the current version in development is the focus of the push, and issues are then ported to service packs for the product's current shipping version. The rationale for targeting the code base in development is that making large changes to a nonshipping product is much easier than doing it once a product has shipped. We take this approach if a product has not shipped yet (more time for regression tests) and because we cannot readily change the default settings for a released product. We could ask users if they want the default changed at service pack installation time, but we can't question them too often (to avoid confusion and annoyance at the onslaught of dialog boxes).

Understanding that a security push is only the start of a cultural change within Microsoft is important, and we make this obvious to all team members. The days of creating designs and code that emphasize features over security are over, and what was acceptable just three or four years ago is simply not tolerable today. We have discovered that security push education followed with an intensive period of design, code, test, and documentation scrutiny instills good habits. More importantly, a critical mass of people at Microsoft really understands security, which affects those around them.

However, having one or more security pushes does not in itself ensure the delivery of secure software: the purpose of a security push is to start process change. To this end, we are modifying our development processes:

- Mandatory security training for new employees
- Threat modeling as a prerequisite for the design complete milestone
- Security design, code, and test guidelines
- Time built into the schedule for ongoing security reviews
- Time built into the schedule for a security push focused on code reviews and testing
- Ongoing security education

In the near term, most people outside Microsoft will perceive the effects of the Windows Security Push by observing the Windows .NET Server 2003 product. The security push changed the product in four fundamental ways:

- It found and removed previously undiscovered vulnerabilities.
- It changed the ways in which the product is implemented so as to reduce the likelihood that undiscovered vulnerabilities remain.
- It changed the product's design so that fewer features and services are enabled by default; if vulnerabilities remain in these features or services, they will be of less concern to the majority of customers who use the product.
- It changed the product's design so that the effect of remaining vulnerabilities will shrink by introducing a degree of defense in depth into the product's design.

Overall, we believe that these changes created a product that customers will perceive is significantly more secure when compared with its predecessors. For articles and developer material about some of the findings from the push, see the following Web pages: http://msdn.microsoft.com/library/en-us/dnsecure/html/strsafe.asp; http://msdn.microsoft.com/library/en-us/dncode/html/secure01192002.asp; and http://msdn.microsoft.com/library/en-us/dncode/html/secure08192002.asp.

We are in the process of measuring the security push's effectiveness in achieving two objectives: improving the product's security and improving the software development processes. Informally, we have observed that the Windows Security Push improved both the security of the Windows code base (by removing vulnerabilities, adding defense in depth capabilities, and reducing attack surface) and the effectiveness of Windows developers at producing secure code. On that basis, we consider the investment that the security push required to have been worthwhile. □

References

1. M. Howard and D. LeBlanc, *Writing Secure Code*, Microsoft Press, 2001.
2. E. Amoroso, *Fundamentals of Computer Security Technology*, Prentice Hall, 1994.

Michael Howard *is a senior security program manager in the Secure Windows Initiative at Microsoft. He coauthored* Writing Secure Code *and* Designing Secure Web-based Applications for Microsoft Windows 2000*, both from Microsoft Press. Contact him at mikehow@microsoft.com.*

Steve Lipner *is the director of security assurance at Microsoft. He has over 30 years' experience with computer and network security research and product development. He is a member of the US National Computer Systems Security and Privacy Advisory Board. Contact him at slipner@microsoft.com.*