Understanding Software Patching

Developing and deploying patches is an increasingly important part of the software development process.

Software patching is an increasingly important aspect of today's computing environment as the volume, complexity, and number of configurations under which a piece of software runs have grown considerably. Software architects and developers do everything they can to build secure, bug-free software products. To ensure quality, development teams leverage all the tools and techniques at their disposal. For example, software architects incorporate security threat models into their designs, and QA engineers develop automated test suites that include sophisticated code-defect analysis tools.

Even under ideal conditions, however, problems always arise. Most software will be used for many years in an ever-changing user environment. This can place new compatibility demands on software and introduce new security vulnerabilities not originally envisioned. Whatever their source, problems can be found in any piece of software and must be addressed with patches.

This article describes the software patching lifecycle and presents some of the challenges involved in creating a patch, deploying it, and monitoring its effectiveness from the perspective of both the developer of a software product and the user who needs to apply the patch. While readers are likely familiar with many of the issues addressed here, my intention is to provide an overview of patching that will help frame one's thinking when tackling these problems rather than to suggest specific

JOSEPH DADZIE, MICROSOFT

solutions to the problems themselves. The primary focus is on security patches, but the issues discussed are equally applicable to nonsecurity-related defects in any software.

IDENTIFYING THE PROBLEM

Security-related patches are common in the software development world. In many cases, security researchers and hackers find vulnerabilities missed during the development cycle, but software vendors find some themselves after the product ships. In the best case, those who find a problem will notify the vendor immediately, before publicly announcing the vulnerability. Other times they do not, however. In some cases they even post exploit code publicly prior to availability of a fix, thereby greatly increasing the risk to users of the affected component.

Regardless of the source of the vulnerability, the software vendor has a responsibility to research the issue and, if valid, produce a patch to address the problem and distribute it as widely as possible.

DEVELOPING THE PATCH

Once the vulnerability is identified, the patch development process begins. The primary goal of that process is to release a patch that addresses the identified issue (and related issues) and introduces no functionality regressions in a relatively short time, depending on how the vulnerability was found and whether "sample" exploit code is publicly available.

UNDERSTANDING THE PROBLEM

Developing a patch requires a thorough understanding of the problem beyond what the finder reported. In some cases, the vulnerability is a simple code flaw that may be easy to fix. In other cases, it may be a much more difficult architectural issue or a problem with how two components interact.

Even simple code flaws may not be easy to fix, however. For example, the flaw may be in one component, but the exploit occurs somewhere else. The WebDAV issue fixed in Microsoft Security Bulletin MS03-007 was an example of one of these. While the exploit happened in WebDAV, the actual problem occurred in a kernel function used by more than 6,000 other components in the operating system. A simple code flaw is no longer easy to fix when you have 6,000 callers. Some of those callers may actually be relying on what you have now determined to be flawed behavior and may have problems if you fix it.

Some of the key questions to complete an understanding of the scope of the problem include:

- Does the problem impact just a single version, multiple versions, or out-of-date versions of the software component?
- Is the component a platform component that lots of other applications depend on?
- Do other components in the software have similar vulnerabilities that may need to be addressed?
- Is the vulnerable software component redistributed with other software?

DESIGNING/CREATING THE FIX

Creating a robust fix in the shortest time possible is not always as straightforward as it may seem, even after the problem is understood. For example, finding developers who have expertise on the vulnerable code is not always possible. This may be simply because the original developer of the component no longer works on it or because that developer has left the company. Or it may be because the component has not been touched in years; hence, there is no active owner to work on it. A few years ago, Microsoft issued a fix for a vulnerability in Gopher that was difficult to get right because at the time there was no active owner for the Gopher protocol.

Designing the fix becomes more involved if the vulnerability is in a common code path or the component is embedded in other software. The recent JPG vulnerability (MS04-028) was an extreme case of a common code-path and redistributed component problem. Not only was the JPG parser used by many different components both in the operating system and in Microsoft and third-party products, many products shipped with their own versions of the JPG parser. It was therefore an immense challenge to design the right fix that would cover all the cases and versions out there.

TRADE-OFFS AND DECISIONS

Creating the right fix will likely involve trade-offs. Careful attention must be given to security and functionality issues, with an understanding that loss in functionality will most likely be perceived as a negative by customers.

A common topic of debate is the scope and release schedule for a fix. For example, if there is a proof-of-concept exploit code out publicly, a decision may be made to short-circuit the normal patch development processes to get a fix out sooner. This may result in a less stable fix than one that had gone through the normal process. For example, in fall 2004 exploit code was posted for previously unknown vulnerabilities in Internet Explorer on public mailing lists on the Internet. Within days real exploits were used to compromise thousands of systems. Microsoft had to circumvent the normal patch procedure to release a fix to prevent the exploits.

Another decision point is how to create a fix that minimizes the chances of potential loss of functionality in the affected software. This is especially critical if the affected component is a platform that is relied on by lots of other software. For example, patches to Internet Explorer, which provides both a browsing experience and a development environment, have to be done carefully given the sheer volume of Web sites and applications that rely on it today.

If a problem impacts multiple versions of a component, a single fix for all versions will be preferable, but not necessarily possible, because of different code bases, different compilers used, different locales, etc. For example, in 1999, Microsoft announced a patch for a vulnerability in Internet Information Services 3.0 and 4.0 (MS99-022) that affected only those systems whose default language was a double-byte language, such as Korean, Japanese, or Chinese. In this case, one patch fixed the problem in all double-byte languages. In other cases, a fix may inadvertently not work on some localized versions of the operating system. This was the case with MS03-045, wherein a patch had installation problems on machines running some non-English versions of the operating system, whereas it worked on others.

If the vulnerability is in a common code path or a component with APIs, then a very broad compatibility test pass will be required to make sure that there is as little loss of functionality as possible. The compatibility testing gets even more complicated if the affected component is embedded in software not developed by your company. These other component owners need to be engaged in the process to ensure that their customers are also protected when the fix is released. A good example of this challenge was the vulnerability discussed in MS02-039 that was exploited by the Slammer virus in 2003.

In any case, the compatibility testing on a patch is never as broad as that for a major or even a minor software release. The reason is that patches are always released to fix some extant problem, and as such there is pressure to get the patches out expeditiously to protect customers. The problem is that if the test pass is cut down too much, the stability of the patch will suffer, resulting in reduced customer confidence in the patches, and, of course, other potential problems including data loss. The larger and more complicated the software and the more versions of it available, the longer the test pass must be.

Developers should not consider the patch development process complete until threat models, design specs, test plans, and code analysis tools are updated to ensure that similar types of vulnerabilities are caught when new software is being developed.

CREATING A DEPLOYABLE PACKAGE

After the development of a fix, it must be packaged in a form that is easily deployable and installable by end users, typically through automated patch management solutions. Patch management systems can be provided by vendors, such as Microsoft's Windows Update, Apple's Software Update service, and the Red Hat Network; or by stand-alone tools such as Microsoft's Systems Management Server.

Given the reality that any product may have more than one vulnerability to be addressed over its lifetime, processes for ensuring that patches do not undo/conflict with previous patches in the same component must be instituted and a standard packaging format must be selected for all patches for a given product. The packaging format often depends on the installation technology used for the software. Examples of packaging formats include Windows Installer for Windows and RPM (Red Hat Package Management) for Linux.

Once the packaging format is determined, the finished packages must be easily deployable to a wide range of users and configurations. Regardless of format, a patch must be in a self-contained package that is easily identified by name, target application or operating system, processor architecture, and language locale. In addition, the patch's relationship with previously released patches must be described and called out.

No matter how clearly described, patch packages can't be easily deployed unless they can be easily installed. This means that packages must support silent installation through the command line of the target operating system. The silent installation options must be the same for all patches for the same application to ease the administrative cost of incorporating the patch into patch management solutions. Once initiated, the install process, if possible, should not require restart of the operating system or affected application. To accomplish

Understanding Software Patching

this, developers need to consider important technical questions, such as whether or not services/daemons can be restarted automatically and in-memory code can be patched while running. If a restart cannot be avoided, the package must provide an option for that restart to be suppressed or controlled by patch management tools to minimize the number of restarts if more than one patch is being applied to a system.

As with the software products themselves, installing patches doesn't always go smoothly. Therefore, it's important for a patch installation to log its activity to a common log so the patch application process can be tracked for reporting and debugging of problems if they occur. Patches should also support uninstall or rollback. This is especially important if problems occur with the patch. The uninstall or rollback process must account for patches that may have been installed out of order with respect to when they were released.

Finally, patches must be protected from tampering and their integrity verifiable through digital signatures, hashes, or checksums. Digital signatures are preferred since the source of the patch can be also be verified.

For components that are redistributed with other software, creation of a deployable package is a more involved process. Can multiple versions of the component reside on the same system? Were these versions shipped using the same installer technology or different ones? If they were different, a strategy needs to be devised to create the right packages for the different programs.

The deployable package is what is used to do the regression and compatibility testing of the patch across affected systems (including different processor architectures and locales).

DISTRIBUTING THE PATCH

Once a deployable package has been verified to fix the issue and has passed all regression and compatibility testing, the package is ready for wide distribution. Distributing the patch is not only a matter of making the package that fixes the vulnerability available to customers. Detailed information on the vulnerability, affected components (including file versions), dependent applications, and changed files may need to be provided along with the patch. Customers need to be notified of the availability of the patch so they can download and deploy it on their systems as quickly as possible. It is also important to provide information on the severity of the problem, the urgency of the fix, and potential mitigations in case a customer cannot immediately deploy the patch. It is common for malicious users to use the detailed description of the vulnerability, potential attack vectors, as well as the patched and unpatched binaries themselves, to create exploits. A competent attacker may be able to do so in minutes. Therefore, a decision has to be made about how much information is necessary for end users and corporate administrators to properly deploy the patch without giving attackers superfluous information they can use to attack innocent users.

It's important that patches get distributed quickly and efficiently to end users, especially when the patch addresses a critical security vulnerability. To accomplish this, patch distribution systems/services must include an automated method for easily distributing patches to end users' machines. This patch distribution service must have adequate service capacity and bandwidth availability to handle the often high number of concurrent users attempting to download the patch. It must also contain provisions for ensuring service if capacity is exceeded or if the service itself is under attack. This is crucial when worms exploiting a vulnerable system attempt to attack the distribution service. An example is the release of the Blaster virus in 2003, as the worm attempted to perpetrate a denial-of-service attack on the Windows Update servers (windowsupdate.com) that were delivering the patch.

Developers of patches must consider not only the bandwidth of their distribution system as a whole, but also the specific bandwidth required to download an individual patch. This is particularly important for dial-up users. The patch may be several megabytes, whereas the attack payload may be only a few hundred bytes, making computers on dial-up links particularly vulnerable.

As mentioned previously, the potential for denial-of-service attacks complicates the patch distribution process. Another challenge to consider is whether a virus can find out which machines need the patch and prevent these machines from ever downloading the patch or exploit them before they do. Patch distribution services must account for this in their designs.

A related problem is ensuring users that they are talking to a legitimate service and that the patches they download are, indeed, what they claim to be and not some malicious virus. Certificates are often used for this purpose, but decisions need to be made about whether the service trusts all certificates installed on the local machine or if it trusts only specific certificates (including checking to see if they have been revoked).

In addition to the bandwidth and security issues already discussed, several other issues should be considered when distributing patches to end users, including:

- How are dependencies between patches described and handled?
- How does the patch distribution system handle situations where the user uninstalls the patch? Does it try to force it down? Does it send a status report that users are uninstalling it?
- Can administrators who have deployed the system in their environment get data on which machines have been patched and which have not?

MONITORING STATUS OF THE PATCH

Once a patch is made available for download, the status of the patch download and install must be tracked by the service or management tool that is distributing it, with the primary goal being to achieve a high install rate of the patch on the potentially affected computers. The tracking also helps determine if the patch is actually reaching all (or a large percentage of) the potentially affected users, if it's being successfully installed, and if it's introducing any compatibility issues with other applications.

The task of tracking/monitoring patch status is made easier if a patch distribution service is being used as the primary distribution mechanism. Some of the key items to keep in mind are:

Patch download status. Are customers successfully downloading the patch? If not, is there a problem with your download server scalability?

Patch install reliability. Are downloaded patches being installed with no errors? Do you understand what errors are being encountered and why? Are the failures within a defined failure threshold? Do users know that the failure is occurring, or is it silent?

Restarts. Are restarts being done for patches that require them? A patch that requires a system restart is not necessarily successfully installed if the restart has not occurred.

Install rate. How many users have installed the patch within a day, week, two weeks, and month of release? Why are users not installing the patch? Is there a compatibility issue that needs to be addressed?

Patch priority. Does the monitoring system take into account the fact that some patches supersede others?

PATCH RECIPIENT PERSPECTIVE

I've thus far provided an overview of the patch development and deployment process from the patch provider's perspective. I would be remiss in my duties, however, if I did not touch upon patch management from the recipient's perspective. My discussion of this perspective should not only help patch recipients think through the salient issues involved in applying patches, but should also be of use to patch providers as a way of stretching their own thinking to include that of their intended users. I encourage you to take a look at George Brandman's article, "Patching the Enterprise," in this issue for a more thorough discussion of managing vendor-issued patches within a large enterprise. For now, I'll provide a brief outline of some of the key issues and questions that customers deal with when patching their installed software products. As with the earlier discussion on patch development/deployment, I tend to focus on security-related patches, but most of these considerations apply equally to nonsecurity-related defects.

Customers running software that requires a patch have different issues to consider once a patch is available from the vendor. The first thing is for customers to know that a patch is in fact available for software they are using. Is there a systemic way that they get notified or do they find out on the news? Once customers are aware of a patch, they need to decide whether or not to deploy it.

For home or end users without IT staff, the vendor's patch management system described earlier should make it extremely easy for them to install the patch without having to become experts. For corporate customers, however, deciding whether to deploy a patch is more involved. The first question to answer is: Is the patch relevant to their network? If so, vulnerable systems must be identified and it must be determined whether firewalls and other network perimeter protection technologies provide mitigations for the exploit of the vulnerability.

IDENTIFYING VULNERABLE MACHINES

Identifying vulnerable machines and instances of the software that may need to be patched can be simple or difficult depending on a number of factors:

- Can the patch management tools being used (*if* they're being used) automatically detect vulnerable systems, or does the IT staff have to write custom scripts based on the information provided with the patch? Do the detection methods work across multiple language locales?
- Is there a fairly large percentage of mobile users? How often do they log in to the network? The methodology for detecting vulnerable systems must account for the

fact that not all machines will be on a network all the time, and therefore strategies need to be implemented to identify these vulnerable machines as soon as they come onto the network.

- Is there a method to determine the vulnerability of machines in remote offices that may or may not have patch management tools or infrastructure? It is common for remote offices to not have either patch management tools installed or administrators who can ensure that machines are patched. What is the strategy for patching those remote offices?
- Do virtual machine/disk images have the vulnerable software on them? Is there an inventory of all such images on the network so a plan can be devised to update them?
- How about development and test networks where new operating systems and applications get installed and uninstalled frequently?

Corporate administrators need to have strategies and patch management processes based on answers to these questions to quickly identify and patch vulnerable machines on their networks.

DEPLOYING AND INSTALLING THE PATCH

Once the corporation has identified vulnerable machines, it is ready to deploy the patch. Like patch developers, corporations must consider many issues when deciding to deploy and install patches on machines within their organizations. Naturally, some of these concerns overlap with those of the software vendors that issue the patches.

The corporation deploying a patch must first determine if there is an exploit out in the wild. An exploit may cause some compatibility testing to be skipped if the patch is critical. It may also cause patches to be force-installed on machines to protect the network.

The corporation must decide if all applications, or only some, need to be tested for compatibility. Mission-critical applications may require extra care to ensure that the business is still functioning after patching. The business should determine if multiple instances of a vulnerable component are running on its machines. If some of these machines are business-critical, they may not be able to be patched except during maintenance windows.

If a business-critical application relies on the vulnerable component, the support contract for the application could be voided if a patch is applied that the vendor has not approved. For patches that apply to server machines, the corporation should consider if these are clustered machines.

Based on these issues, a corporation may decide to employ mitigation on certain critical machines and not apply the patch immediately. This is a risky proposition, but it is up to the IT staff to make that risk assessment. It may decide to schedule downtime outside of the regular maintenance window or prevent network access to vulnerable machines unless a particular patch is installed.

MONITORING PATCH COMPLIANCE

The last phase of the patch process from a customer perspective in a corporate environment is to monitor compliance status to ensure that the network is protected against the vulnerability. This monitoring must be done on an ongoing basis as new unpatched systems and applications get installed on the network. Vulnerable machines may need to be isolated and patched as soon as possible.

THE RESPONSIBILITIES OF PATCHING

The patching cycle is a complicated one with different aspects that need to be thought through carefully during development and use of software. The processes need to be continuously evolved to quickly and effectively keep machines protected and up to date.

The software publisher/patch developer has a responsibility to quickly address vulnerabilities, proactively notify its customers, and provide enough information and tools to allow the patch to be quickly deployed. The end users are responsible for protecting their software assets by deploying patches quickly and effectively without disrupting their businesses. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JOSEPH DADZIE is group program manager for Windows Software Distribution Technology at Microsoft. He has been at Microsoft for more than nine years, focusing on software installation, deployment, and distribution technologies. He is currently responsible for software distribution technologies. He has a B.A. in engineering sciences and B.E. in engineering from Dartmouth College.