Practical Defense in Depth

Michael Howard
Microsoft Corp

As part of its ongoing commitment to Bill Gates' vision of Trustworthy Computing, Microsoft officially adopted important security- and privacy-related disciplines to its software development process. These changes, called the Security Development Lifecycle (SDL) have led to a demonstrable reduction in security vulnerabilities in products such as Microsoft's Windows Vista operating system and its SQL Server 2005 database. The purpose of this article is not to describe the SDL in detail, but to outline some of the practical defensive measurements in use at Microsoft required by the SDL. If Microsoft's SDL is new to you, refer to the page 16 sidebar, "A Brief SDL Overview."

Even though the vulnerability counts have dropped, the number of vulnerabilities is not zero. And, even in my wildest dreams, I do not think we will get to zero. I will explain why shortly.

In the very early days of the SDL, Microsoft focused heavily on removing design and code-level security vulnerabilities; as we progressed, we added processes that help reduce the chance that new vulnerabilities get added to the software.

Examples of implementation requirements in the SDL include:

- Use of code analysis tools on developer's desktops to find security vulnerabilities.
- Removing known insecure functions (such as the C runtime *strcpy* and *strncpy* functions).
- Migrating weak cryptographic algorithms to more robust algorithms (such as Data Encryption Standard to Advanced Encryption Standard, Secure Hash Algorithm (SHA)-1 to SHA-256).

But the SDL is constantly evolving. We update the SDL roughly twice a year so we can keep pace with new vulnerabilities and new security research. We find the continuous improvement to the SDL to be a significant benefit compared to static security certification programs – such as the Common Criteria – which do not evolve so quickly.

Over the last few years, the SDL has been extended to embrace a stronger focus on defense in depth. While some updates to the SDL continue to address design and implementation vulnerabilities, more of today's SDL requirements focus on defense in depth.

The prime driver for this change is a realization that you can never remove all security vulnerabilities from software. The reason this statement is true is simple. Some software you create might be completely secure today, but that could all change tomorrow when security researchers learn (and potentially make public) new classes of vulnerabilities.

Allow me to illustrate the point with a real example.

In October 2003, Microsoft issued a security bulletin, MS03-047 [1] that fixed a cross-site scripting (XSS) vulnerability in the Outlook Web Access (OWA) front end to Microsoft's Exchange 5.5 software. In August 2004, Microsoft issued another bulletin, MS04-026 [2], in the same OWA component that was fixed in MS03-047 to fix an XSS variation called HTTP response splitting. Interestingly, the code fixes were relatively close to one another. So the question that's probably on your mind is, "What happened? How did you guys miss the bug that led to MS04-026?" The answer is simple. At the time we issued MS03-047, the world had not heard of HTTP response splitting vulnerabilities. In theory, the Microsoft Exchange engineers could have scoured the code and fixed every known security bug, but they would probably have missed the vulnerability that led to MS04-026 because nobody knew of that class of vulnerability at the time. So what changed? What led to the security vulnerability? In March 2004, Sanctum (purchased by Watchfire, which has since been purchased by IBM) released a paper entitled "Divide and Conquer" [3] describing a variation of the XSS vulnerability. When the Microsoft engineers fixed the first bug, the second class of bug was unheard of.

> *"Some software you create might be completely secure today, but that could all change tomorrow when security researchers learn (and potentially make public) new classes of vulnerabilities."*

Unfortunately, there were no defense in depth mechanisms in place to protect customers from either of these vulnerabilities, so customers had to apply a security update to protect themselves.

Another example is the integer arithmetic vulnerability [4]. Without going into a detailed explanation, this kind of bug was unheard of 10 years ago, and is a very common security vulnerability today.

The moral of this story is that you can never create totally secure code. There are many reasons why:

- People make mistakes.
- Tools are not perfect.
- Security can have very subtle nuances that only security experts understand, especially with regard to cryptography.
- The Internet is an asymmetric battleground. There are many hidden and skilled attackers who can strike at will, but defenders must be constantly vigilant and never make mistakes.
- Most importantly, we cannot predict future classes of vulnerabilities.

There are two final points that I really wish to stress because these make defense in depth especially critical today:

1. As the security vulnerability landscape evolves, more vulnerability information is moving underground and being used by criminals to attack sensitive systems. The defenders do not know the system is vulnerable, so no security update is available. This is clearly a risk for government systems.
2. Somewhat similar to the first point is that we are seeing more zero-day vulnerabilities. There may be no attacks yet, but there is no update or workaround either.

The rest of this article outlines some

A Brief SDL Overview

The SDL is a set of requirements and recommendations added to an existing software development process to improve security. A side benefit of the SDL is increased robustness since many security vulnerabilities also affect robustness. SDL is all about security improvement, not perfection.

There are two goals to the SDL. The first is to reduce the number of security vulnerabilities in Microsoft products. This is done by removing vulnerabilities from software, or better yet, not adding vulnerabilities to the code from the outset. This reduction in vulnerabilities is achieved through education, tooling, better libraries, and so forth. The second goal is to reduce the severity of any vulnerabilities that are inadvertently left in the product. You can reduce severity by adding defensive mechanisms. The purpose of this article is to explain some of those defenses.

In a nutshell, we teach people to: *Do everything possible to make your product as secure as possible, but assume it will fail.*

A requirement defines an SDL task that must be completed prior to giving the code to customers, and a recommendation defines a best practice that should be considered by the development team. It is not uncommon for a security best practice to start out as a recommendation and then progress to a requirement.

The SDL defines requirements and recommendations for:

- Education.
- Risk assessment.
- Threat modeling.
- Coding.
- Testing.
- Final security review.
- Maintenance.

You can learn more about the SDL in "The Security Development Lifecycle" (Microsoft Press, Howard and Lipner), or at the SDL blog: <http://blogs.msdn.com/sdl>.

of the defense in depth techniques and technologies applied as part of the SDL at Microsoft.

Classes of Defense in Depth Mechanisms

Under the SDL, there are two distinct types of defense in depth mechanisms. We don't call them out explicitly as different classes, but the distinction between the two classes is understood.

The first type of defense is designed to totally stop an attacker from accessing a system or software. This class of defense offers a level of assurance that stops an attacker when the defense is correctly configured and used. If it does not stop an attacker, then the defense has a vulnerability that must be fixed. One example is the ubiquitous firewall. The SDL sets strict requirements on the process for opening an inbound port on the Windows Firewall. Other examples include permissions on objects: a weak permission could render a system insecure. For example, Microsoft issued a security bulletin, MS04-005 [5], for VirtualPC for the Apple Macintosh because of a weak permission on a critical file that led to a symlink-style attack [4].

Strong operating system access controls are an important part of the SDL and critically important to any system; however, a new requirement in the SDL this past year is strong access control mechanisms on database objects. Structured Query Language (SQL) injection vulnerabilities are a well understood vulnerability that can lead to disclosure of sensitive data, data corruption, and, in some cases, system compromise [4]. The industry as a whole has created best practice documentation and tools to help remove these critical bugs, but all it takes is a new attack type or an error to leave a customer open to attack and compromise. The SDL-required defense in depth mechanisms help to mitigate this risk. In short, direct access to the underlying tables is explicitly denied to all but the database administrators, and untrusted access is limited to appropriate database objects such as stored procedures and views. These objects are granted access to the underlying data. This configuration has the effect that if an attacker can bypass the normal defenses against SQL injection, he or she still cannot read the underlying data in the tables. The correct remedy to prevent SQL injection is to build safe SQL queries, but the table-level defense is there solely in case the remedy fails or is implemented incorrectly.

The second type of defense is a set of mechanisms that is designed to slow an attacker down or make an attacker create a different exploit to attack a system. I want to spend most of this article on this subject. At Microsoft, we continue to spend a great deal of time and effort researching, designing, and implementing these defenses: most of them are intended to help mitigate buffer overrun and integer overflow vulnerabilities. A great deal of C and C++ code exists today, and even more is written every day. In a perfect world, people would simply abandon C and C++ in favor of safer programming languages such as C# or Java, but in our imperfect world, C and C++ are often the correct tools for the job. Again, in a perfect world, C and C++ developers would write secure code, but in our imperfect world this is not always possible; however, it is important that C and C++ developers take advantage of defense in depth mechanisms. The SDL mandates a number of important C and C++ defenses, including:

- Address randomization.
- Stack-based buffer overrun detection.
- Heap corruption detection.
- Pointer protection.
- No-execute (often called NX or W^X).
- Service failure restart policy.

Note that none of these defenses actually remove vulnerabilities, nor do they magically make software more secure. What they do is turn a potential code execution exploit into a denial-of-service bug because these defenses will simply fail the application if they detect an anomalous condition. If an application crashes, it also gives the attacker fewer opportunities to re-attempt an attack.

I do not intend to cover each of these in deep technical detail. The reader is urged to e-mail the author or refer to the references for further information [6, 7].

Address Randomization

There is nothing attackers love more than a predictable system since it makes building reliable exploits easier. Reliable exploits are harder to detect because they usually execute correctly and do not crash or alert the system operators to nefarious acts. Windows Vista and Windows Server 2008 (and later) offer image randomization, stack randomization, and heap randomization. Image randomization relocates the entire operating system into one of 256 possible configurations on each reboot. By default, non-operating system images are not randomized, and third-party components must opt into image randomization using the /DYNAMICBASE linker option in Visual C++ 2005 Service Pack 1 and later. Some versions of GNU's C Compiler (GCC) and some versions of Linux and Berkeley Software Distribution (BSD)-based systems also support image randomization by using the -pie compiler option.

Windows Vista and Windows Server

2008 and later also support stack randomization; when a thread is started, the thread's stack is offset by up to 32 pages (4 kilobytes on a 32-bit central processing unit). Again, this option is available by linking with /DYNAMICBASE.

Finally, Windows Vista and Windows Server 2008 (and later) support heap randomization, meaning that when an application allocates dynamic memory from the system heap, the operating system offsets the start of the heap by a random amount. Heap randomization is enabled by default in Windows Vista and Windows Server 2008.

Together, image, stack, and heap randomization can seriously hinder an attacker dealing with the lack of predictability. Usually the sign of a failed attack is a crash in the application under attack. This means that system administrators should really pay attention to applications that crash, as crashes may not just be the sign of a coding bug but a sign of a security bug under attack.

Stack-Based Buffer Overrun Detection

Stack-based buffer overrun detection is available in Microsoft Visual C++ 2003 and later through the /GS compiler switch. It works by adding a random number into a function's stack frame at call time and when the function returns code inserted by the compiler, it verifies that the random number has not changed. If it has changed, then the application crashes because a stack-based buffer overrun has been detected. At this point, we can no longer trust the integrity of the data or the application. Some versions of the GCC offer a similar defense by using -fstack-protector.

This defense could require the attacker to build a specific attack to circumvent the defense. A good example of how this helped protect customer is the Blaster Worm. On Windows Server 2003 (but not Windows 2000 or Windows XP), the vulnerable component was compiled with /GS. The malicious Blaster payload was not aware of the /GS defense, so it crashed Windows Server 2003 machines rather than infecting them with the Blaster Worm.

Heap Corruption Detection

Heap corruption detection is an operating system defense available in Windows Vista and Windows Server 2008 (and later). It is similar in principle to stack-based buffer overrun detection but detects heap metadata corruption. Again, if the heap is corrupted, the operating system can shut down the application, reducing the chance that an attacker will re-attempt a failed attack.

Pointer Protection

C and C++ pointers are an attack vector; if an attacker can overwrite a long-lived pointer in memory, he or she can potentially compromise a computer by writing arbitrary data at a predictable location. Windows includes functions that encode (XOR) a pointer with a random value, and this operation must be reversed successfully in order to get the valid pointer value. In other words, if an attacker attempts to overwrite a pointer, he or she must overwrite it with a value that survives the unencoding operation. Clearly, this is not impossible, but it is another defense the attacker must overcome. The application programming interfaces in Windows that perform these operations are:

- EncodePointer and DecodePointer.
- EncodeSystemPointer and Decode SystemPointer.

Note that GLIBC v2.5 (and later) have a similar defense, but it's mainly used inside GLIBC itself to protect *setjmp* pointers. The functions, defined in *sysdep.h* are PTR_MANGLE and PTR_DEMANGLE.

No-Execute (Often Called W^X)

Microsoft calls no-execute data execution prevention (DEP). This defense marks pages of memory as *Writeable* or *Executable*, but not both. Essentially, this makes it very hard for an attacker to run malicious code out of a writeable memory segment. Most CPUs today support this capability. It is by no means a perfect defense and DEP requires randomization to be effective at stopping a class of attacks known as *return-to-libc* [8].

In Windows, you can link with /NXCOMPAT to opt-in for DEP.

Some versions of BSD and Linux support W^X also, but the compiler and operating system support is not consistent across platforms.

Service Failure Restart Policy

A final defense in Windows Vista and Windows Server 2008 was very hard to implement without sacrificing reliability. Windows uses many services, akin to Unix daemons, to perform critical system tasks. Services are usually long-lived processes that start when a system starts. In many cases, administrators want a crashed service to simply restart. This policy gives better uptime to customers. This is great for reliability, but it can be terrible for security because it means that an attacker can keep trying his attacks over and over until they succeed. The ability to retry is especially important in a system that implements a great deal of randomization, such as Windows Vista and Windows Server 2008.

Windows offers the ability to define a policy that restarts a failed service no more than a certain number of times or on a certain schedule. For example, an administrator could define a policy that will restart a process 10 times within 24 hours, and after that no longer allow it to restart unless an administrator physically restarts the service. It is also possible to restart a process indefinitely. But we don't want to give attackers the ability to re-try their attacks indefinitely, so we tightened up the restart policy for many highly exposed system services.

For example, in Windows Server 2008, many services, including the Network Access Protection Agent, are set to restart twice, and then no longer restart. In other words, the attacker has two shots. With all the randomization in place in Windows today, this makes the attacker's job much more difficult.

The Question of Least Privilege

You may have noticed that I have not mentioned least privilege as a defense, and I left it out on purpose. Clearly, least privilege is an important defense, but it is necessarily an imperfect wall because many products have had and will continue to have local escalation of privilege vulnerabilities. Also, least privilege does not mitigate many information disclosure vulnerabilities. Malicious code running as a normal user, rather than an administrator, can still access data accessible by the user, and that data could include sensitive data such as passwords, encryption keys, personal financial information, and e-mail. Within the SDL we think of least privilege as very important, but we also recognize that on a normal user's computer, it can be hard to enforce the security boundary and have a usable system. A good example of this is running mobile code through a Web browser. At some point, a user will probably visit a Web site that requires a Java applet, a Flash file, or perhaps some multimedia experience that will require some mobile code. Installing this code is a trusted operation, so the user must elevate to an account that can install the code. The process of elevating can lead to weaknesses in a pure least privilege environment. Of course, it is possible to utterly lock a system down in such a way that it is very

difficult for a user to elevate at all; for some installations processing sensitive information, this is the right answer.

Summary

Writing code that is perfectly secure in the long term is not possible; new attack types appear almost weekly. But it is imperative that systems offer a degree of protection, even in the face of new classes of attacks and design and coding vulnerabilities. This means that software development organizations should spend a great deal of time thinking about defense in depth mechanisms, as well as focusing on "getting the code right." A simple mantra to consider is, "Your code will fail – now what?"

The problem is exacerbated by zero-day vulnerabilities, and vulnerability research moving underground to be used for criminal purposes.

If there is one lesson we can all learn, it is this: Defense in depth is just as important as following good security coding and design practices, because you will never get the product totally secure.

If there is a second lesson, it is that you must use as many defense in depth mechanisms as possible and they must be enabled by default because defense in depth is most useful in the face of an attack that takes advantage of a vulnerability that is not publicly known.

We have implemented the defenses listed above in various Microsoft products. I would urge you to take advantage of these defenses if you build on the Microsoft platform; if you use other products, understand what defense in depth mechanisms they offer and use them.◆

References

1. "Vulnerability in Exchange Server 5.5 Outlook Web Access Could Allow Cross-Site Scripting Attack." Online posting. 12 Apr. 2004 <www.microsoft.com/technet/security/bulletin/ms03-047.mspx>.
2. "Vulnerability in Exchange Server 5.5 Outlook Web Access Could Allow Cross-Site Scripting and Spoofing Attacks." Online posting. 10 Aug. 2004 <www.microsoft.com/technet/security/bulletin/ms04-026.mspx>.
3. Klein, Amit. "Divide and Conquer." HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Mar. 2004 <www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf>.
4. Howard, Michael, David LeBlanc, and John Viega. 19 Deadly Sins of Software Security. Emeryville, CA: McGraw Hill, 2005.
5. "Vulnerability in Virtual PC for Mac Could Allow Privilege Elevation." Online posting. 10 Feb. 2004 <www.microsoft.com/technet/security/bulletin/ms04-005.mspx>.
6. Howard, Michael, and Matt Thomlinson. "Windows Vista ISV Security." Microsoft Developer Network. Apr. 2007 <http://msdn2.microsoft.com/en-us/library/bb430720.aspx>.
7. Howard, Michael. "Protecting Your Code with Visual C++ Defenses." MSDN Magazine. Mar. 2008 <http://msdn2.microsoft.com/en-us/magazine/cc337897.aspx>.
8. McDonald, John. "Defeating Solaris/SPARC Non-Executable Stack Protection." Online posting. 2 Mar. 1999 <www.ouah.org/non-exec-stack-sol.html>.

About the Author

Michael Howard is a principal security program manager on the Trustworthy Computing Group's Security Engineering team at Microsoft, where he is responsible for managing secure design, programming, and testing techniques across the company. Howard is an architect of the SDL, a process for improving the security of Microsoft's software. He began his career with Microsoft in 1992 at the company's New Zealand office, working for the first two years with Windows and compilers on the Product Support Services team, and then with Microsoft Consulting Services, where he provided security infrastructure support to customers and assisted in the design of custom solutions and development of software. In 1997, he moved to the United States to work on Internet Information Services, Microsoft's next-generation Web server, before moving to his current role in 2000. Howard is a Certified Information Systems Security Professional and a frequent speaker at security-related conferences. He regularly publishes articles on security design and is the co-author of six security books, including the award-winning "Writing Secure Code," "19 Deadly Sins of Software Security," "The Security Development Lifecycle," and his most recent release, "Writing Secure Code for Windows Vista."

E-mail: mikehow@microsoft.com