Editors: Richard Ford, rford@se.fit.edu
Michael A. Howard, mikehow@microsoft.com

A Process for Performing Security Code Reviews

No one really likes reviewing source code for security vulnerabilities; it's slow, tedious, and mind-numbingly boring. Yet, code review is a critical component of shipping secure software to customers. Neglecting it isn't an option.

Michael A. Howard
Microsoft

I get to review quite a bit of code—not as much as I used to, but enough to keep me busy helping teams at Microsoft. Sometimes people just want my take on small snippets of perhaps 100 lines of code, and other times I get hundreds of thousands of lines.

People often ask how I review code for security vulnerabilities when faced with a massive amount to review. At a very high level, my process is simple:

- Make sure you know what you're doing.
- Prioritize.
- Review the code.

Although my approach might not work for you, I've fine-tuned it over the years based on comments and lessons from people I consider much better than me at reviewing code. Let's look at each step in more detail.

Make sure you know what you're doing

This sounds obvious, but unless you know what a security bug is, you have no chance whatsoever of finding them while reviewing code. So the first-order task is to become educated on the issues. For some good books on the subject, see the "Further reading" sidebar.

Of course, there's no replacement for experience, and so we at Microsoft often pair up more senior code reviewers with apprentices to help pass on code review wisdom. Learning from other people's experience is a critical component of education, and, frankly, it works better for some classes of people than sitting in a lecture-style environment.

If you're new to this field, you'll then want to review old security vulnerabilities. Check out SecurityFocus.com and sign up for Bugtraq. Spend time understanding the core issues and proposed remedies in the discussions. Also, look at security vulnerabilities you and your colleagues have confronted.

Prioritize

How do you prioritize a code review effort when you have, say, 1,000 files to review? Various tactics that gravitate around attack surface and potential bug density can provide a good place to start. Code compiled into higher attack surface software requires deeper review than code in lower attack surface components. This isn't to say that you don't need to review lower attack surface software; I'm simply talking about priorities. We can use the following heuristics to determine code review priority:

- *Old code.* Older code tends to have more security vulnerabilities than new code because newer code often reflects a better understanding of security issues. The definition of *old* is hard to quantify, but you should review in depth any code you consider "legacy."
- *Code that runs by default.* Attackers often go after installed code that runs by default. Therefore, such code must be better quality, and hence reviewed earlier and deeper than code that doesn't execute by default. Code running by default increases an application's attack surface, which is a product of all code accessible to attackers.[1]
- *Code that runs in elevated context.* Code that runs in elevated identities—Local System in Windows or root in *nix, for example—must be of higher quality and requires earlier and deeper review because code identity is another component of attack surface.
- *Anonymously accessible code.* Another attack-surface element, code that anonymous users can access should be reviewed in greater depth than code that only valid users and administrators can access.
- *Code listening on a globally accessible network interface.* Code that listens by default on a network, especially the Internet, is obviously open to substantial risk and must be reviewed in depth for security vulnerabilities.
- *Code written in C/C++/assembly language.* Because these languages have direct access to memory, buffer-manipulation vulnerabilities within the code can lead to buffer overruns, which often lead to malicious code execution. With this in mind, you should review code written in these

languages in depth for buffer-overrun vulnerabilities. Of course, other languages have other catastrophic vulnerabilities, such as SQL-injection vulnerabilities in Java, PHP, C#, or Perl.

- *Code with a history of vulnerabilities.* Code that's had numerous past security vulnerabilities probably has many more unless someone has made a directed effort to remove them from it.
- *Code that handles sensitive data.* You must review code that handles personal, financial, or confidential data to ensure that it doesn't disclose the data to untrusted users through potential weaknesses.
- *Complex code.* There is no good metric to define *complex code*, and although some research at Microsoft suggests that cyclomatic complexity[2] might be a good indicator of potential bug density, I know of no metric that's a great indicator of potential security-bug density. Nonetheless, complex code often has bugs and some of them could well be security vulnerabilities.
- *Code that changes frequently.* Code that exhibits churn often sees new bugs introduced. Again, not all of these will be security vulnerabilities, but given a stable set of code that's updated only twice in a year versus code that changes every day, the latter will probably have more vulnerabilities in it.

Another prioritization technique is to get an estimate of the code's bug density. The capture–recapture method provides one good way to do this. Although it was developed to determine wildlife populations,[3] we can also use it to estimate bug density. At the highest level, this involves the following process:

- Have a small team (TeamA) of perhaps just two or three people review some code for security vulnerabilities (BugsA).
- Have a second team (TeamB) review the same code, also looking for security vulnerabilities (BugsB).
- Note the quantity of vulnerabilities found by both teams (BugsAB).
- Estimate the number of vulnerabilities in the code using the formula:

  BugsA / Estimate = BugsAB / BugsB.

For example, if the first team finds 10 security vulnerabilities and the second team finds 12, with four in common, the capture–recapture method estimates 30 vulnerabilities in the code. In other words, the review team probably has a lot more work to do. Note that, although a general bug count is interesting and useful, we find no evidence at Microsoft of a correlation between general bug quantity and security bug quantity.

Review the code

The next step is where the work really begins. Reviewing code involves three steps:

- Rerun all available code-analysis tools.
- Look for common vulnerability patterns.
- Dig deep into risky code.

Let's look at each in detail, starting broad but shallow, digging deeper into risky areas, and finally getting narrow but very deep.

Rerun all available tools

Most software shops have source-code analysis tools. They range from simple, noisy grep-like tools that look for potential vulnerabilities to full-fledged static-analysis tools that perform data flow analysis on the code under inspection. Even the highest compiler warning levels (such as /W4 in Visual C++ or –Wall in gcc) can indicate areas of code that require more scrutiny. Rerun all tools, and note every warning or error—even those you'd normally reject as noisy. Some warnings could actually be errors, or at least hide real errors. Warnings from multiple tools can indicate code that needs more security review.

Look for common vulnerability patterns

The next step is to review for some of the most common and nefarious bug types. Most notable among these are integer arithmetic issues, buffer overruns, cryptographic issues, SQL injection, and cross-site scripting (XSS).

The best way to illustrate the process of searching for these bug types is to use a simple decision graph. But first, it's important to understand one simple rule: you must always know what the attacker controls. If the attacker controls nothing, there's no security bug; if the

Further reading

The following books are good places to start with the issues surrounding security vulnerabilities:

- M. Howard and D. LeBlanc, *Writing Secure Code*, 2nd ed., (Microsoft Press, 2002)
- M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security*, (McGraw-Hill, 2005)
- G. Hoglund and G. McGraw, *Exploiting Software*, (Addison-Wesley, 2004)
- H. Thompson and S. Chase, *The Software Vulnerability Guide* (Charles River Media, 2005)
- J. Whittaker and H. Thompson, *How to Break Software Security*, (Addison-Wesley, 2003)
- M. Andrews and J. Whittaker, *How to Break Web Software*, (Addison-Wesley, 2006)

It's best to read two or three of these books thoroughly—especially those that cover the area in which you're developing solutions or software products.

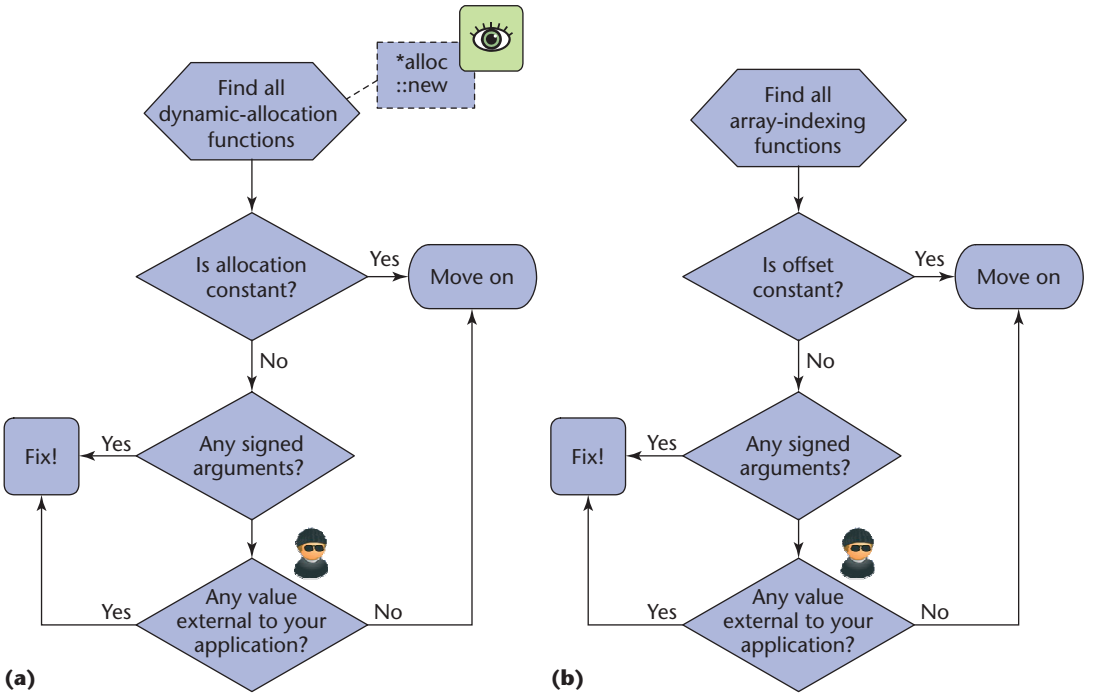

Figure 1. Reviewing for integer overflow issues. (a) Dynamic memory-allocation functions and (b) array indexing code must be correctly bounds checked to make sure there is no arithmetic errors that could lead to security vulnerabilities.

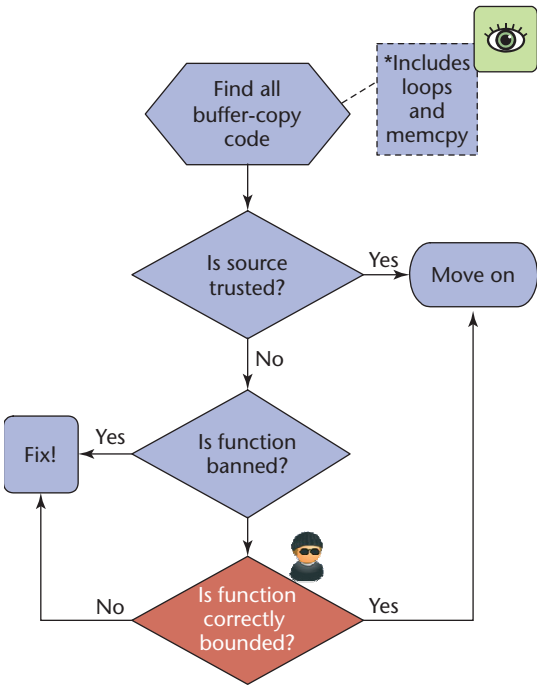

Figure 2. Reviewing for buffer overflows. This chart shows the high-level flow from identifying common coding constructs to determining whether they're used securely.

attacker controls a great deal of data used in the code, the potential for a security bug skyrockets. Consider, for example, the following code, which shows a buffer overrun:

```
char t[3];
t[3] = 0;
```

Is this code a security bug? The answer is no. It is most certainly a bug, but the attacker controls nothing. If the attacker controlled, say, the array index, then it would be a first-class security bug.

Now let's look at the common vulnerabilities. I don't offer remedies here, but the books listed in the sidebar have great advice on how to fix them.

Integer arithmetic vulnerabilities.

The two flow charts in Figure 1 show the process of reviewing code for integer arithmetic issues. Figure 1a illustrates the process for finding vulnerabilities in memory allocation, and 1b shows how to find them during array indexing.

When determining if your code has any dynamic memory-allocation functions, you can start by searching for `*alloc` and calls to `operator::new`. That said, you'll still have to locate those rare cases in which memory-allocation functions don't have `*alloc` in their names. For example, the integer overflow that led to Microsoft bulletin MS05-051 was in a function named `GetMemSpace`.[4]

Buffer-overrun vulnerabilities.

Good tools can find some classes of buffer overruns, but not all of them, so manual review is still important—particularly because many tools focus on "dangerous functions" rather than data origin. The first step is to identify all code that copies buffers and isn't restricted to `strcpy`. You can start by locating `while()` or `for(;;)` loops or "safe" func-

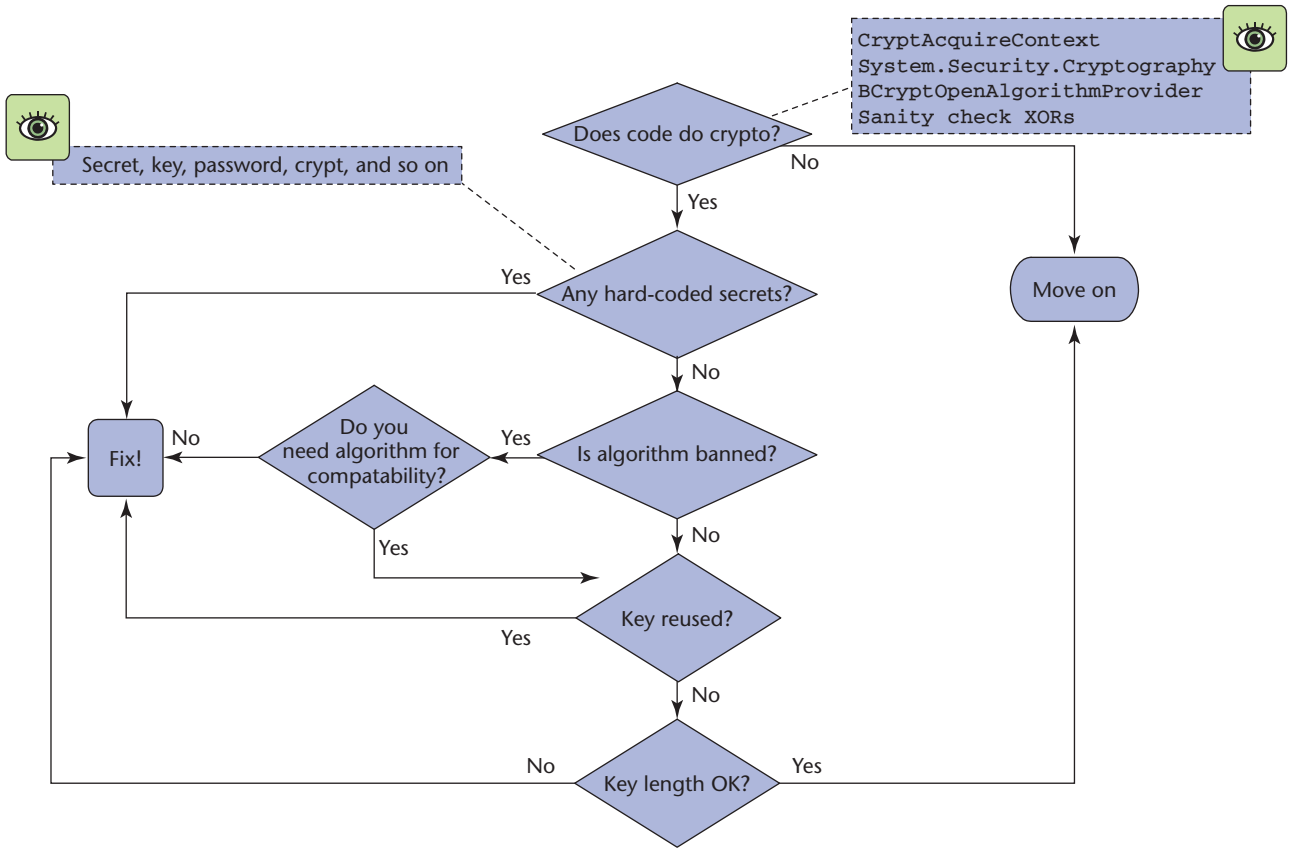

Figure 3. Reviewing for crypto issues. This chart shows the steps for determining whether your code uses cryptographic constructs and, if it does, how to restrict the code to using more robust algorithms and key sizes.

tions such as `memcpy`, which programmers commonly use to copy buffers. The bug that led to the Blaster worm was a simple `while` loop.

Given the prevalence of buffer-overrun exploits that arise from them, Microsoft has actually banned many C runtime library functions for new code including:

- the `strcpy`, `wcscpy` family,
- the `strcat`, `wcscat` family,
- the `strncpy`, `wcsncat` family,
- the `sprintf`, `wsprintf` family,
- the `snprintf`, `wnsprintf` family,
- the `vsprintf`, `wvsprintf` family,
- the `_alloca` family,
- the `strtok` family,
- the `scanf` family, and
- the `gets` family.

In *The Security Development Lifecycle*, Steve Lipner and I include a full list of banned functions along with a header file, named banned.h.[5]

Note the last block in the flow chart in Figure 2: Is function correctly bounded? I can't stress enough how important this is. I've seen code in which the developer replaced a potentially insecure function, such as `strcpy`, with an ostensibly secure function such as `strncpy`, but got the buffer size wrong. This is the error that led to the June 2006 Symantec remote-management-stack buffer overflow advisory; you can see the coding error on the eEye Web site (www.eeye.com/html/research/advisories/AD20060612.html).

Cryptographic vulnerabilities. The chart in Figure 3 shows the process for finding cryptographic weaknesses in your code. If your code uses cryptographic algorithms, you need to make sure the code has a base level of "hygiene." That means no hard-coded secrets, such as passwords or cryptographic keys, and no banned algorithms. Microsoft has banned several algorithms for new code including:

- DES (key size is too small)
- MD4, MD5 (broken)
- SHA-1 (showing signs of weakness)
- RC4 (unless reviewed by a cryptographer)

In some cases you have no choice but to use a banned algorithm because an industry standard requires it. For example, digest authentication[6] requires MD5. But we avoid them whenever possible, or at least

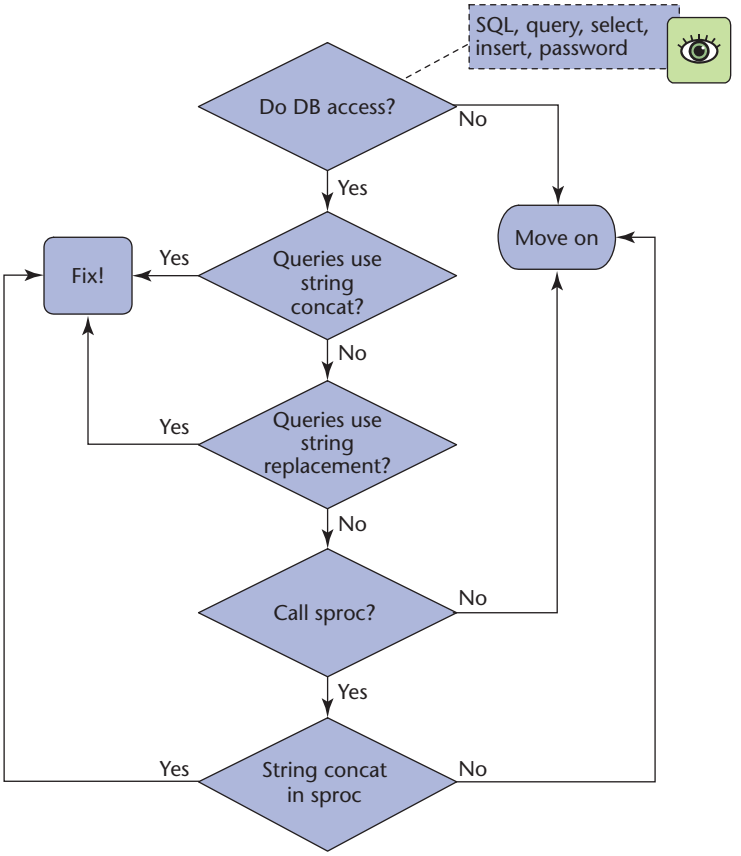

Figure 4. Reviewing for SQL-injection issues. First you must determine whether your application uses a SQL database or not, if it does you must perform appropriate SQL statement construction hygiene.

limit their use to code that isn't security-related.

SQL Injection vulnerabilities. SQL injection vulnerabilities are a prime attack method for compromising sensitive data. The "we use a firewall" argument simply doesn't help most of the time, so you must make sure your code is as free from SQL-injection vulnerabilities as possible.

Cross-site scripting vulnerabilities. XSS issues can lead to disclosure of a Web-user's private data, and some XSS-based worms, such as the MySpace XSS worm, have been seen in the wild. The part of the flow chart in Figure 5 that requires explanation is with regard to defenses. You can use many defense-in-depth tricks to help mitigate any XSS vulnerabilities you might miss. These include HTML-encoding all output that's derived from untrusted input, and marking cookies as HttpOnly[7] to help mitigate cookie-stealing XSS attacks.

Dig deep into risky code

The process I described earlier finds only low-hanging vulnerabilities. The last stage of the code-review process is to dig very deeply into risky code. But be warned that this phase is slow going. Most people tire quickly of reviewing code, so I suggest setting aside only two hours to review code and having no more than two such sessions a day.

The first step is to assemble a very small team of usually no more than three or four people:

- the code author, whose job is just to answer questions, not to play tour guide,
- a subject-matter expert who is knowledgeable about the code, but isn't the code author, and
- a note-taker.

Including more than four generally slows the process down as people tend to argue and get sidetracked on unrelated issues.

The next step is to identify the riskiest components—you should be able to turn to the threat model for this list, or else you can determine potential risk in a less formal manner by applying the heuristics I defined earlier. However you identify the components, be sure to manually review all anonymously accessible code that's exposed to the Internet.

Once you document all these components, build a code review schedule. For each component under review, determine where data enters the system—this might be a network call via `recv()` or `recvfrom()`—and then trace the data from that point forward, questioning the nature of the data and how it's used in the code. Consider the following questions when manipulating the data:

- Are there logic or *off-by-one* errors (for example, '>' vs. '>=' or '||' vs. '&&')?
- Is the data correctly validated?
- Are buffer lengths constrained correctly?
- Are integer values range-checked correctly?
- Are pointers validated?
- Can the code become inefficient (for example, $O(N^{\geq 2})$) due to some malformed data (for example, a hash table look-up becomes a list look-up[8])?
- Are errors handled correctly?

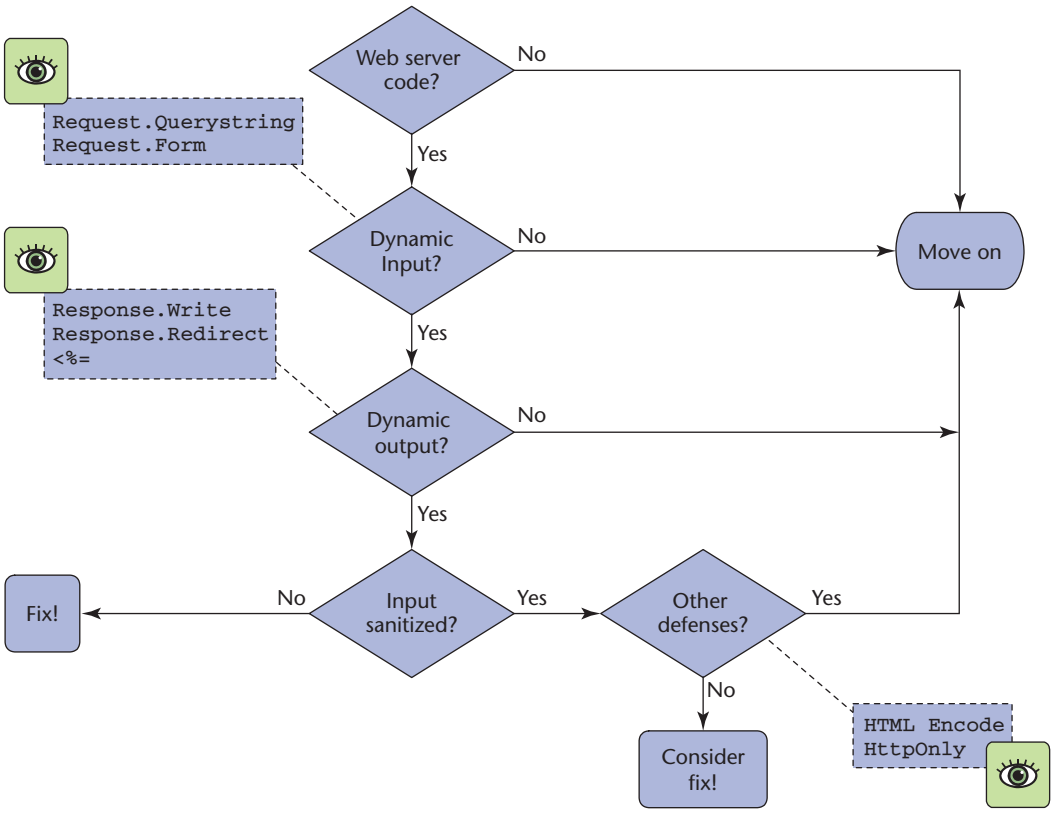

Figure 5. Reviewing for cross-site scripting (XSS) issues. This chart shows the high-level steps required to identify common XSS issues in Web server code.

You can stop the code review when you get to a point in the code that correctly validates the input. This might occur in more than one point, so make sure to review all such code.

This process will not find all security vulnerabilities in your code, but it's an effective method for scanning large amounts of code quickly for common issues and reviewing risky code in greater depth. A good source-code editor that lets you quickly browse, perform lookups, and hop around the call structure is critical. And never lose track of my greatest advice to you here: always understand what the bad guy controls. ▢

References

1. M. Howard, "Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users," http://msdn.microsoft.com/msdnmag/issues/04/11/AttackSurface/default.aspx.
2. T.J. McCabe and A.H. Watson, "Software Complexity," *Crosstalk, J. Defense Software Engineering*, vol. 7, no. 12, 1994, pp. 5–9.
3. S.C. Amstrup et al., eds., *Handbook of Capture-Recapture Analysis*, Princeton University Press, 2006.
4. "Vulnerabilities in MSDTC and COM+ Could Allow Remote Code Execution," Microsoft Security Bulletin MS05-051, 11 Oct. 2005; www.microsoft.com/technet/security/Bulletin/MS05-051.mspx.
5. M. Howard and S.B. Lipner, *The Security Development Lifecycle*, Microsoft Press, 2006.
6. J. Franks et al., *HTTP Authentication: Basic and Digest Access Authentication*, Internet Eng. Task Force RFC 2617, June 1999; www.ietf.org/rfc/rfc2617.txt.
7. Mitigating Cross-site Scripting with HTTP-only Cookies; http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp.
8. S.A. Crosby and D.S. Wallach, *Denial of Service via Algorithmic Complexity Attacks*, tech. report, Aug. 2003; www.cs.rice.edu/~scrosby/hash/.

Michael A. Howard *is a senior security program manager in the Security Engineering group at Microsoft. He is the coauthor of* Security Development Lifecycle, Writing Secure Code, *and* 19 Deadly Sins of Software Security *(Microsoft Press). He focuses on secure design, development and testing policies and best practice. Contact him at mikehow@microsoft.com.*