

# Improving Software Assurance<sup>1</sup>

Robert J. Ellison and Carol Woody

Software Engineering Institute

Software assurance objectives include reducing the likelihood of vulnerabilities such as those on a [Top 25 Common Weakness Enumeration](#) (CWE) list and increasing confidence that the system behaves as expected. Practitioners should understand where to look, what to look for, and how to demonstrate improvement. To delve deeper into software assurance, the [BSI website](#) provides a wealth of information to aid in tying security into all development activities.

## Where to Look

Initial discussions of system security often include firewalls, authentication issues such as strong passwords, or authorization mechanisms such as role-based access controls, but the defects that typically enable an attack are not in the code that directly implements security.

An increasing number of attacks occur in the code that implements the functionality. The [CWE](#) classifies security bugs and documents the design and coding weaknesses associated with attacks. The CWE list is dominated by errors in the functional software. As the operating system and network security vulnerabilities were reduced, applications became the next attack target. Application security has often been ignored, in part because of the faulty assumption that firewalls and other perimeter defenses could protect the functional code. The problem is further compounded as application developers without specific security training are typically unaware of the ways that their software that meets functional requirements could be compromised. Security software is usually subject to an independent security assessment that considers the development history as well as the design and operation. There is no equivalent effort applied to the security of the functional code.

## What to Look For

Two analysis techniques described below, attack surface and threat modeling, can be useful in identifying critical locations where vulnerabilities are likely to be found.

### ***Attack Surface***

An approach to managing the scope of the analysis arose from pragmatic considerations. Howard in 2003 observed that attacks on Windows systems typically exploited a short list of features such as open ports, services running by default, services running as SYSTEM, dynamically generated web pages, enabled accounts, enabled accounts in admin group, enabled guest accounts, and weak access controls [Howard 2003a]. Instead of counting bugs in the code or the number of vulnerability reports, Howard proposed to measure the attack opportunities, a weighted sum of the exploitable features. An attack-surface metric is used to compare multiple versions or configurations of a single system. It cannot be used to compare different systems.

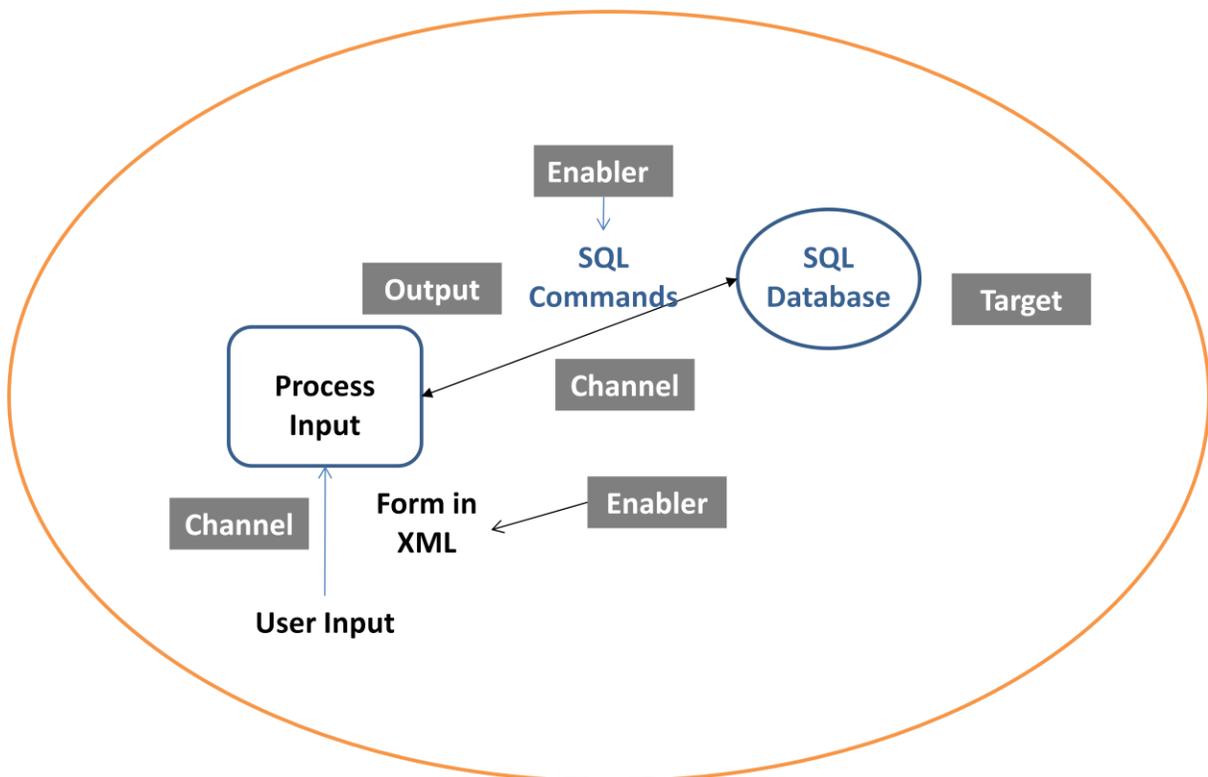
Howard's intuitive description of an attack surface led to a more formal definition with the following dimensions: [Howard 2003b]

---

<sup>1</sup> Material presented at the 43rd Annual Hawaii International Conference on System Sciences, 5-8 January 2010, Koloa, Kauai, Hawaii.

- targets—data resources or processes desired by attacker (A process could be web browser, web server, firewall, mail client, database server, etc.)
- enablers—the other processes and data resources used by attacker (e.g. web services, mail client, or having JavaScript or ActiveX enabled. Mechanisms such as JavaScript or ActiveX give the attacker a way to execute their own code.)
- channels and protocols (inputs and outputs)—used by attacker to obtain control over targets
- access rights—control is subject to constraints imposed by access rights

A simple attack surface is shown in Figure 1 for an application that accepts user input and inserts that input into a database query.



**Figure 1: Attack Surface Example**

There is not an accepted way to calculate a numeric value for the area of an attack surface from these factors. The attack surface area in Howard's calculation is the sum of independent contributions from a set of channel types, a set of process-target types, a set of data target types, and a set of process enablers, where each type is given a weight and all are subject to the constraints of the access rights.

Let's consider the example in Figure 1. The attack target is a database with the attack surface weight determined by the type of data. A database with credit card or other customer financial information would have a high weight. The channels and protocols are significant contributors to the attack surface for this example. A customer input function where the input value is determined by the selection of an icon has a low risk of exploits. Whereas a function that accepts a user-filled form with entries that are used to query a database should be given a high weight since such types of channels have a history of exploits that gave attackers access to confidential information.

An attack surface supports consideration of software assurance risk in several ways.

- A system with more targets, more enablers, more channels, or more generous access rights provides more opportunities to the attacker.
- Feature usage of product influences the attack surface for that acquirer. The attack surface can be used to compare the opportunities for attacks when usage changes.
- An attack surface helps to focus the analysis on the code that has to be trusted. A reduced attack surface also reduces the code that has to be evaluated for threats and vulnerabilities.
- For each element of a documented attack surface, the known weaknesses (CWE) and attack patterns can be used to mitigate the risks.
- The attack surface supports deployment as it helps to identify the attack opportunities that could require additional mitigation beyond that provided by the product.

### ***Threat Modeling***

The creation of an attack surface helps to focus analysis but does not identify the security risks and possible mitigations for the functional components. Threat modeling is a systematic approach to identify security risks in the software and rank them based on level of concern [Howard 2006, Swiderski 2004].

Threat modeling constructs a high-level application model (e.g. data flow diagrams), a list of assets that require protection, and a list of risks. For this discussion, we assume that it also produces a list of mitigations that is not necessarily a requirement in Microsoft's characterization of the practice.

A detailed walk-through of a data flow considers the deployed configuration and expected usage, identifies external dependencies such as required services, analyzes the interfaces to other components (inputs and outputs), and documents security assumptions and trust boundaries, such as the security control points [NIST 2007].

The problem often occurs when we compose systems with incompatible assumptions into systems of systems. For example, a classic security problem with legacy systems is that they were often designed under the assumption of operating in a trusted and isolated environment. That assumption is violated if such legacy systems are used as a backend for a web-based public-facing front-end. Threat modeling a multi-system data flow can identify such assumption mismatches.

Consider the attack-surface example shown in Figure 1. Threat modeling analyzes the data flow associated with that figure. If the user submits an ID value of 48983, then the output from the input routine is likely a database command, such as

```
SELECT ID name salary FROM EMPLOYEES
WHERE ID=48983
```

in response to which the server returns

```
48983 Sally Middleton $74,210.
```

For properly formatted input, the data flow will successfully execute and returns the desired data or an error if the ID is not found.

Threat modeling analyzes how this data flow could be compromised by malformed input. In this example, user input, the ID, has been used to create the database query. Threat modeling draws on known attack patterns and vulnerabilities such as CWE-89 (Improper Sanitization of Special Elements used in an SQL Command.) For the database example,

malformed data could include database commands. The symbol | is interpreted by a SQL database server as a logical OR. Input of 48983 | 1 = 1 would download information for all employees as the selection criteria ID = 48983 or 1 = 1 is always true. This technique has frequently been used in attacks that illegally downloaded credit card data. A number of the 2009 Top 25 items apply to this example.

- CWE-20: Improper Input Validation
- CWE-89: Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')
- CWE-116: Improper Output Encoding or Escaping

CWE-20 and CWE-89 refer to errors with processing the input. As noted by Howard, CWE-116 is not really a bug except by omission. Howard also notes that a number of failures in input processing could have been prevented by proper encoding or escaping [Howard 2009]. With MySQL if the input in the example is escaped with quotes, "48983 | 1= 1" or with the built-in MySQL escape function, the embedded MySQL commands are not interpreted and the escaped output would have generated a MySQL processing error.

A data flow where user input is part of a command that will be executed by another process automatically raises a "red flag" for a security-knowledgeable developer given the extensive history of software defects associated with such data flow. Table 1 shows an example where user input is used to create the full name that will be used to access a file in folder A. If that input can include file system commands such as "../" then the user may be able to access files outside of the intended folder, which is represented by CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal').

<b><i>Input</i></b>	<b><i>Command</i></b>	<b><i>Comment</i></b>
Costs	C:\\A\\Costs	Access file <i>Costs</i>
..\B\Costs	C:\\A\\..\B\Costs	Changed folder

**Table 1: File System Command**

Inputs for the examples in Table 2 are URLs that an attacker has convinced a user to submit. Web server vulnerabilities associated with the processing of those URLs can adversely affect the user and the web server. A number of the entries on the 2009 and 2010 Top 25 CWE lists correspond to weaknesses in web applications. Such CWE's include

- CWE-20: Improper Input Validation [2009]
- CWE-79: Failure to Preserve Web Page Structure (cross-site scripting) [2009, 2010]
- CWE-98: Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion') [2010]
- CWE-116: Improper Output Encoding or Escaping [2009]
- CWE-209: Information Exposure Through an Error Message [2010]
- CWE-306: Missing Authentication for Critical Function [2010]
- CWE-311: Missing Encryption of Sensitive Data [2010]
- CWE-352: Cross-Site Request Forgery [2009, 2010]

- CWE-434: Unrestricted Upload of File with Dangerous Type [2010]
- CWE-601: URL Redirection to Untrusted Site ('Open Redirect') [2010]
- CWE-602: Client-Side Enforcement of Server-Side Security [2009]
- CWE-732: Incorrect Permission Assignment for Critical Resource [2010]
- CWE-754: Improper Check for Unusual or Exceptional Conditions [2010]
- CWE-807: Reliance on Untrusted Inputs in a Security Decision [2010]

The attacker's objective with the first example is to have the bogus site displayed and hopefully inherit the user's trust. Such a URL should be rejected by the trusted site's web server software, but there are numerous examples where server software accepts such URLs and loads the bogus site. Web server execution of embedded JavaScript in the second example can pass user data on that server to the attacker.

<i>Command</i>	<i>Comment</i>
<a href="http://trustedsite/.../bogus_site">http://trustedsite/.../bogus_site</a>	Redirection may lead user to trust bogus site.
<a href="http://trustedsite.../JavaScript">http://trustedsite.../JavaScript</a>	JavaScript embedded in the URL may give attacker access to user data on trusted site.

**Table 2: Web Server URLs**

## How to Demonstrate Improvement

Threat modeling is never complete and cannot guarantee that functional code is free of security-related defects. It is based on current knowledge. New attack techniques may be applicable to code that was once thought to be secure.

Today, more than 25 large-scale software security initiatives are underway in organizations as diverse as multi-national banks, independent software vendors, the U.S. Air Force, and embedded systems manufacturers. The Software Assurance Forum for Excellence in Code (SAFECode), an industry-leading non-profit organization that focuses on the advancement of effective software assurance methods, published a report on secure software development [Simpson 2008]. In 2009, the first version of The Building Security In Maturity Model (BSIMM) was published [McGraw 2009]. BSIMM was created from a survey of nine organizations with active software security initiatives that the authors considered to be the most advanced. The nine organizations were drawn from three verticals: four financial services firms, three independent software vendors, and two technology firms. Those companies among the nine who agreed to be identified include Adobe, The Depository Trust & Clearing Corporation (DTCC), EMC, Google, Microsoft, QUALCOMM, and Wells Fargo.

The improved software assurance that results from defect identification and mitigation associated with threat modeling or equivalent risk analysis techniques reduces the overall risk for those using the software component. The benefits of threat modeling for the developer are documented in [Howard 2006]. Those benefits include requiring development staff to review the functional architecture and design from a security perspective, contributing to the reduction of the attack surface, and providing guidance for code reviews and security testing. The threat model identifies the components that require an in-depth review and the kinds of malformed input that might expose defects. For the database example, security and penetration testing should include SQL injection

and may be able to use commercial tools designed to find SQL-injection vulnerabilities. See the BSI [Security Testing](#) content area.

Increased attention on secure application software components has influenced security testing practices. All of the organizations contributing to the Build Security In Maturity Model do penetration testing, but there is increasing use of fuzz testing. Fuzz testing creates malformed data and observes application behavior when such data is consumed. Fuzz testing does not generate purely random data but typically uses knowledge about the protocol, file format, and content values. An unexpected application failure due to malformed input is a reliability bug and possibly a security bug. Fuzz testing has been used effectively by attackers to find vulnerabilities. For example, in 2009, a fuzz testing tool generated XML-formatted data that revealed an exploitable defect in widely used XML libraries [Codenomicon 2009]. At Microsoft, about 20 to 25 percent of security bugs in code not subject to secure coding practices are found via fuzz testing. For example, an application that reads a file is tested with 100,000 automatically generated malformed entries. An application that fails unexpectedly is retested after repairs against a different stream of malformed files. The files that generated failures are archived and used to test new versions.

## Summary

Only a subset of any given Top 25 CWE bug list will be applicable for a given system, and a focus on just the Top 25 risks could miss serious risks for that system. Where a Top 25 weakness is applicable, a comparison an organization's current software development practices with recommended mitigations for such weaknesses can provide a warning that practices should be improved. A Top 25 CWE bug should not be a surprise to a system's developers.

A software assurance objective is to incorporate the identification and mitigation of likely design, coding, and technology-specific weaknesses into the development life cycle. This note provided a sketch of two practices that support that objective. Mitigations of items on a Top 25 CWE list are usually linked to detailed design or coding practices but weaknesses mitigations are also associated with risk analysis, requirements, architecture, and testing. The [BSI website](#) provides a foundation for a full life-cycle context for security improvement.

## References

- [Codenomicon 2009] For more information, read Codenomicon's [press release](#).
- [Howard 2003a] Howard, Michael. [Fending Off Future Attacks by Reducing Attack Surface](#). (2003).
- [Howard 2003b] Howard, Michael, Pincus, Jon, & Wing, Jeannette. [Measuring Relative Attack Surfaces](#). (2003).
- [Howard 2006] Howard, Michael & Lipner, Steve. *The Security Development Lifecycle*. Microsoft Press, 2006.
- [Howard 2009] Howard, Michael. "Improving Software Security by Eliminating the CWE Top 25 Vulnerabilities." *IEEE Computer*. (June/July 2009): 68-71.
- [McGraw 2009] McGraw, Gary, Chess, Brian, & Miguez, Sammy. [The Building Security in Maturity Model](#) (2009).

- [Simpson 2008] Simpson, Stacy, ed. [\*Fundamental Practices for Secure Software Development: A Guide to the Most Effective Secure Development Practices in Use Today.\*](#) (2008).
- [Swiderski 2004] Swiderski, Frank & Snyder, Window. *Threat Modeling.* Microsoft Press, 2004.