

The Seven Habits of Highly Insecure Software

By Herbert H. Thompson, Ph.D.

Summary: Severe functional bugs usually have pretty overt symptoms: an application crash, corrupt data, and screen corruption. Security bugs, though, usually have more subtle symptoms and habits. This article discusses the most common and difficult-to-notice symptoms of insecure software to help you track down these bugs during testing.

In his book, *The Seven Habits of Highly Effective People*, Stephen R. Covey identifies behaviors from some exceptional people who have proven themselves the most highly effective and productive. Covey observed these people, studied their habits, and taught these techniques and skills to others. In this article, we have taken a similar approach but with much more unsavory subjects of study: insecure software. My goal was to identify symptoms of insecure software, telltale signs that the application is behaving in a way that could create security vulnerabilities. By understanding how insecure software behaves, we can better detect and remove these types of bugs in our own software.

With this in mind, we scoured bug databases for the most malevolent and destructive security bugs ever to infest a released binary. We sifted through thousands of bug reports, incident reports, and advisories. Common characteristics started to emerge. Habits like writing out temporary files with secure data, leaving unprotected ports open, and foolishly trusting third-party components, just to name a few. The result was our (James Whittaker's and my) book, *How to Break Software Security*, (Addison-Wesley, 2003), which outlines prescriptive testing techniques to expose security bugs in software. This article summarizes some of the insecure habits we've found, and identifies the symptoms of the most common security vulnerabilities and how they can be detected and avoided.

Habit # 1: Poorly Constrained Input

By far, the number one cause of security vulnerabilities in software stems from the failure to properly constrain input. The most infamous security vulnerability resulting from this habit is the buffer overflow. Buffer overflows happen when application developers use languages (like C and C++) that allow them to allocate a fixed amount of memory to hold some user-supplied data. This usually doesn't present a problem when input is properly constrained or when input strings are of the length that developers expected. When data makes it past these checks, though, it can overwrite space in memory reserved for other data, and in some cases force commands in the input string to be executed. Other unconstrained input can cause problems, too, like escape characters, reserved words, commands, and SQL (Structured Query Language) statements.

Habit # 2: Temporary Files

Usually we think of the file system as a place to store persistent data; information that will still be there when the power is shut off. Applications, though, also write out temporary files—files that store data only for a short period and then are deleted. Temporary files can create major security holes when sensitive data is exposed. Common (inappropriate) uses of temp files include user credentials (passwords), unencrypted but sensitive information (CD-keys), among others.

Habit # 3: Securing Only the Most Common Access Route

How many ways could you open a text document in Windows? You could double-click on the file in Windows Explorer; or open your favorite text editor, and type the file name in the open dialog; or type the file name into an Internet Explorer window. The truth is, if you put your mind to it, you could think of at least a dozen ways to open that file. Now imagine implementing some security control on that document. You would have to think of every possible access route to the

document, and chances are, you're likely to miss a few.

Developers fall into this dilemma too. When requirements change, or when a new application version is being developed, security controls are often "added-on" to an application. Also, when a security bug is reported, developers may patch the application to fix the particular input sequence reported and still leave other, underused access routes unprotected. The result: the reappearance of supposedly fixed bugs or alternate access routes that bypass security mechanisms.

Habit # 4: Insecure Defaults

We are all guilty of the mortal sin of clicking "Next" or "Finish" on an installation wizard without reading the details and just accept recommended configurations. But is it a sin? The application's developers and testers know more about the application than we do, so it seems natural not to worry about awkward installation options and just accept defaults. Most users think this way and I can't say that I blame them. So what does this mean for security-conscious testers? It means that we need to ensure security out of the box. We have to make sure that default values err on the side of security, and that insecure configurations are appropriately explained to users.

Habit # 5: Trust of the Registry and File System Data

When developers read information from the registry, they trust that the values are accurate and haven't been tampered with maliciously. This is especially true if their code wrote those values to the registry in the first place. One of the most extreme vulnerabilities is when sensitive data, such as passwords, is stored unprotected in the registry. We have found that passwords, configuration options, CD keys, and other sensitive data are often stored unencrypted in the registry—ripe for the reading.

Habit # 6: Unconstrained Application Logic

It's pretty clear that we need to examine individual functions to make sure that they are secure. If a feature used in a Web browser is not supposed to allow the reading of any file except a cookie, then there's a pretty good chance that a test case was run to verify that. Features are not likely to be as well constrained when they are combined or when commands are executed in a loop. Constraining loops can be an exquisitely difficult programming task. Many denial of service attacks are made possible by getting some benign function (such as one that writes out a cookie) to execute over and over again and consume system resources.

Habit # 7: Poor Security Checks with Respect to Time

The ideal situation is that every time sensitive operations are performed, checks are made to ensure they will succeed securely. If too much time lapses between time-of-check and time-of-use, then the possibility for the attacker to get in the middle of such a transaction must be considered. It is the old "bait and switch" con applied to computing: Bait the application with legitimate information, and then switch that information with illegitimate data before the application notices. Using these seven habits as a guideline for your software project will help ensure a successful outcome. There's no such thing as 100 percent bug free software. Our goal, however, is to get as close as possible.

About the Author Herbert H. Thompson is Director of Security Technology at Security Innovation LLC. He earned his Ph.D. in Mathematics from the Florida Institute of Technology. Herbert is co-author (with James Whittaker) of *How to Break Software Security: Effective Techniques for Security Testing* (Addison-Wesley, 2003), and is the author of numerous papers on software security and testing.

Source for this Document STQe-Letter 20 Aug 2003.