

Software Assumptions Lead to Preventable Errors

Undocumented assumptions are often the cause of serious software system failure. Thus, to reduce such failures, developers must become better at discovering and documenting their assumptions. In the March/April issue of this magazine,

original CGI specification calls for each CGI process to be executed for each Web call. FastCGI changes this model to allow a CGI to run as a long-running process and avoid the startup costs of a new process. Many CGI applications in which memory leaks weren't considered to be a problem (because they had a short lifetime) rapidly consumed all the available memory on a busy Web server when they were converted to FastCGI.

Like the memory leak example, global state and thread-unsafe storage are a major cause of problems in software. We need only look back to the mid 1990s to see how API developers' failure to document interface thread safety led to untold numbers of problems when multithreaded designs became commonplace. Lest we place too much blame on developers, their designs often made assumptions that they didn't even realize were assumptions.

In the security space, an early common assumption was that software would run on a single-user standalone system, so security objects such as files and IPC mechanisms wouldn't need specific permissions, nor would they need to be configured with the notion of multiple users. Developers expected that anyone who could log in ought to have the full use of all the PC's files. If developers assume that their software will have exclusive use of these primitives, they might also assume that resource contention won't occur. TOCTOU (time-of-check-to-time-of-use) race-condition errors are only

ANDY
STEINGRUEBL
PayPal

GUNNAR
PETERSON
Arctec Group

the Secure Systems department reviewed some specific, significant software failures that could be viewed as failures to document critical assumptions.¹ In this article, we focus on common categories of assumptions in software, discuss methods for recognizing when developers are making them, and recommend techniques for documenting them, which offers value in and of itself.

The Problem's Scope

Consider a recent project that required you to design and build a software (or even a physical) object. How many design decisions did you base on evidence versus local custom? Often, the unwritten ways in which the organization's development and security staff "know" how projects typically deploy and operate fundamentally affects design decisions. Because these local customs are implicit, they're rarely questioned, even as they drive project requirements. Although these implicit requirements—assumptions—might accurately reflect a given organization's state of affairs at a given time, this rarely remains true for long. An undocumented assumption could become invalid if, for example, an architect

decides to integrate new clients in a client-server system with a service-oriented architecture (SOA), replace Fortran with Java, or swap a single-core PC with a multicore one.

The issue of undocumented assumptions isn't always one of insufficient documentation but of awareness. Developers don't always recognize that they're even making assumptions, so we must focus on devising techniques that focus on areas where assumptions can occur and assisting developers so that they can methodically examine them.

Examples of Software Assumptions

Developers often make assumptions about runtime environments or specific use cases for the software they create—for example, assumptions that a piece of software will run for a short time can lead developers to assume memory leaks won't matter. When the software is incorporated into a long-running process, this assumption comes back to bite the future user. A common example of this type of assumption occurred in the late 1990s when certain Perl or C/C++ applications written as CGIs were converted to FastCGI. The

one manifestation of this type of assumption.³

Still more security assumptions take for granted that a given interface is “private” and can only be called by trusted parties or those inside some form of perimeter. With the rise of SOA and Web services, newly integrated systems frequently violate security assumptions because code that used to exist only on a trusted code path gets reused in an integration with a less-trusted party. This can happen, for example, when a company’s mainframe system, whose security model is comprised of mainframes and terminals (which are typically isolated), and a resource-centric authorization model such as Resource and Admission Control Functions (RACF) are integrated using SOA and Web services. The “back end” data and functionality generally stay largely the same, but the front end “green screen” terminal is replaced with a Web browser (out of the mainframe’s control), so the mainframe access control system makes decisions with partial information and partial control. The new design puts Web services in front of the mainframe, meaning it’s no longer isolated from other systems. Furthermore, because the interaction between users and the system is no longer completely RACF-mediated, control of the mainframe’s data and functions moves upstream to the user.

Documenting and Handling Assumptions

Fortunately for the software developer, several techniques exist for both documenting and analyzing assumptions. These include design-time techniques, such as dataflow diagrams and their security-conscious cousin, the threat model; implementation-time technical techniques, such as code annotation and design-by-contract languages; and runtime techniques, such as system call permissions,

and operating system features that allow for fine-grained definition of capabilities.

Security is context-sensitive: something that’s appropriate for a convenience store in the hills of Kentucky differs from what’s needed in downtown Miami. The techniques described here provide a way to document assumptions about design time, deployment time, and runtime in a software context.

Design-Time Assumptions

One of the best techniques for documenting design-time assumptions is the old-fashioned functional specification. Explicit requirements in a specification are a strong way of ensuring that all use cases are covered and that no behavior is left undocumented. Dataflow diagrams are another useful mechanism for documenting assumptions. If developers are careful to document data flows, then none of them should be hidden, and thus no hidden dependencies on other components should be expected. Threat modeling is a more formal mechanism for documenting dataflow security assumptions as well as security trust boundaries, inputs and outputs, and component interactions. For a good introduction to real-world threat modeling including tooling, we recommend Microsoft’s information on the subject as a starting point (<http://msdn.microsoft.com/en-us/security/aa570413.aspx>). Although Microsoft’s model had traditionally been heavily weighted toward doing threat-modeling for all projects, we find that its free documentation on the threat-modeling process along with its free tools provide the best introduction to the process for beginners.

Implementation-Time Assumptions

Documenting assumptions while implementing software can be as simple as providing useful source code comments inline. A step further than code annotations is the

use of design by contract languages such as Eiffel that enforce runtime constraints on pre- and postconditions on function calls, ranges on variables, and strict bounds checking on strings. Although we believe that these strict runtime enforcements result in better assumption documentation, we also admit that they aren’t necessarily useful for the average developer.

Runtime Assumptions

We’ve found that one of the best techniques for documenting runtime assumptions is the software packaging system in use on a given platform. Package managers such as Redhat’s RPM and Debian’s APT capture many of the coarse-grained configuration elements that we should document, such as the runtime user, dependencies on other packages and libraries, start/stop mechanisms, and basic file permissions. Where these package manager configurations end, features such as fine-grained services permissions and capabilities pick up. A more detailed discussion of how Microsoft’s security configuration wizard tool helps make explicit a service’s runtime constraints and environment appears elsewhere ([http://technet.microsoft.com/en-us/library/cc731515\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/cc731515(WS.10).aspx)). Although OpenBSD’s sys-trace and Solaris’s privileges/capabilities model provide roughly equivalent functionality to the features in Windows 2008, Microsoft’s bundling of easy-to-use tools takes a low-level operating system feature and makes it highly likely it will be used both by system administrators and those who create packages for deployment.

Applying the Techniques

The two most effective approaches for applying the techniques we’ve described are policies and checklists. Policies vary in expressability and coverage—ideally, they encapsulate hidden assumptions in

an explicit policy statement, with the advantage being that we can query the policy at a future point when the assumptions no longer

Control assumptions, for example, capture the expected control flow, including function-call ordering and initialization requirements.

Usage assumptions capture how an application expects to be used—will this component be a standalone executable, or will it be called from another interface?

apply. For example, in early Web services development, many Web service providers didn't support WS-Security and SAML, so developers had to dial back to HTTP Basic authentication, which is extremely weak without also using the Secure Sockets Layer (SSL). Simply giving HTTP Basic carte blanche would constitute bad practice, thus a policy statement that binds the use of HTTP Basic to SSL is easy to document, manage, and enforce. Once Web services stacks matured to support WS-Security and SAML more broadly, we could revise this authentication type to reflect more complex policy decisions.

Some of the ideas covered in this article previously appeared elsewhere;² in this earlier work, the authors discussed more techniques for documenting assumptions within code itself, rather than as a comprehensive part of the software development process. Although we believe the taxonomy of assumption categories the authors documented is valid, we aren't convinced that documenting assumptions as statements in code is always the best mechanism.

We've found that it's best to break checklists into two tiers. The first contains the categories of assumptions that can occur, as adapted from a previous work:²

- control,
- environment,
- data,
- usage, and
- convention.

Environment assumptions capture dependencies on other software components, including versions. Data assumptions capture what data the program expects to input and output, including that data's format and type and who's checking it for correctness. Usage assumptions capture how an application expects to be used—will this component be a standalone executable, or will it be called from another interface? Finally, convention assumptions capture the developer's assumptions about interoperability, conventions other developers will use in interpreting their code, statements, and documentation. Each of these assumptions has the potential to cause problems when reality diverges from requirement.

An additional security-centric assumption category relates to the concept of boundaries. On one hand, boundaries are simply control structures, such as access control containers that enforce and constrain relationships among subjects and objects. But in distributed systems, in which security mechanisms span namespaces, technology, and policy domains via federation, capabilities require a subtle distinction. If an identity provider (IdP) in domain A issues a SAML assertion, for example, this assertion represents a set of capabilities that IdP vouches for on behalf of something in domain A. This assertion is then acted on by services X, Y, and Z in domain B, each of which independently evaluates the assertion's bound-

ary and scope for authorization purposes, even though all services in domain B lack authority in domain A.

Although the previously published taxonomy is a useful starting point,² organizations can benefit from adding some specifics for understanding their own pitfalls at a more detailed level. For example, a company could build on the structure of assumption categories with checklist of concerns in their specific system and how they relate. For control,

- Is this application/library/function thread-safe? Does it assume single-threading? Does it have thread-unsafe local storage?
- How many copies of this object will be instantiated at once? Does it matter? What are the bounds?

For environment,

- Who's responsible for authentication in this software system?
- Can we assume that the caller is authenticated and that all data is trustworthy, or are we responsible for performing access control enforcement checks?

For data,

- Who will call this function/class/module? Can we assume they've performed robust safety and correctness checking on the data passed to us, or is that our responsibility?
- What performance assumptions are made about the software? How quickly must it execute? How large a data set will it process?

For usage,

- How will this software be called?
- By whom and when?

Finally, for convention, what character set encoding is assumed?

We can't document a complete

assumptions checklist here, but these workable examples of both security and nonsecurity checklists items should drive home the point that checklists can be a useful tool.

We've talked implicitly about when assumptions matter, but what they almost all have in common is a problem with composition. Although security assumptions in a single component can be problematic, they are also often the culprit in security problems that arise when we try to compose software components. Unstated assumptions about who will validate data or whether a routine is safe for external direct calls or should be mediated by a bounds-checking routine cause us problems when we compose software components with different security expectations. Nonsecurity instances of composition problems happen, for example, when we try to write a new program that accesses a database for read+write, where the database's integrity is maintained by an application rather than constraints in the database itself. When the new application isn't aware of this assumption, it can break the data. A security-specific example occurs when an application composed of components makes assumptions about which components will provide data validation. If a data access layer assumes that it will only be handed data consistent with the data model, it won't be prepared for data coming from an input routine that doesn't filter data appropriately. When the input routine assumes data validation is the data layer's job, this combination of assumptions can result in error conditions such as data corruption or SQL-Injection. Although we certainly don't presume to have solved the composition problem, we believe that the rigorous documenting of implicit

assumptions in our software and systems leads to easier composition, in which side effects, security interactions, and runtime security requirements are made explicit and reasoning about composition made easier.

Disclaimer

The examples of software security or quality issues described herein have been garnered from a variety of different organizations over a long period of time. None of them are reflective of problems encountered at the author's current or former employers. □

References

1. M. Zhivich and R.K. Cunningham, "The Real Cost of Software Errors," *IEEE Security & Privacy*, vol. 7, no. 2, 2009, pp. 87–90.
2. G.A. Lewis et al., "Assumptions Management in Software Development," *Software Eng. Inst.*, Carnegie Mellon Univ., 2004; www.sei.cmu.edu/publications/documents/04.reports/04tn021.html.
3. M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, no. 2, 1996, pp. 131–152.

Andy Steingruebl is a manager in the Information Risk Management Group at PayPal. He manages the secure development program for all PayPal applications, including the Web sites supporting PayPal's 73 million active registered accounts. In addition, he works with the information risk management team on issues of Internet governance, Internet policy, security protocols, and Internet infrastructure. Contact him at asteingruebl@paypal.com or via www.thesecuritypractice.com.

Gunnar Peterson is managing principal of Arctec Group. His research interests include security in distributed systems, and software security. He is a visiting scientist at Carnegie Mellon University's Software Engineering Institute. He maintains an information security blog at <http://1raindrop.typepad.com>.



Call for Articles

IEEE Software seeks practical, readable articles that will appeal to experts and nonexperts alike. The magazine aims to deliver reliable, useful, leading-edge information to software developers, engineers, and managers to help them stay on top of rapid technology change. Topics include requirements, design, construction, tools, project management, process improvement, maintenance, testing, education and training, quality, standards, and more.

Author guidelines: www.computer.org/software/author.htm

Further details: software@computer.org

www.computer.org/software

