OWASP Dev Guide V2.1

OWASP Dev Guide V2.1

Table of Contents

List of Tables

Part I. Front Matter

Version 2.1.1

Cover

A Guide to Building Secure Web Applications and Web Services

2.0 Final Candidate

July 24, 2005

Table of Contents

Frontispiece

Copyright

Editors

The Guide has had several editors, all of whom have contributed immensely as authors, project managers, and editors over the lengthy period of the Guide's gestation.

Table 1. Editor Table

Adrian Wiesmann	Andrew van der Stock	Mark Curphey	Ray Stirbei

Authors and Reviewers

The Guide would not be where it is today without the generous gift of volunteer time and effort from many individuals. If you are one of them, and not on this list, please contact Andrew van der Stock, vanderaj@owasp.org

Table 2. Editor Table

Abraham Kang	Adrian Wiesmann	Alex Russell	Amit Klein	Andrew van der Stock
Christopher Todd	Darrel Grundy	David Endler	Denis Pilipchuk	Dennis Groves
Derek Browne	Eoin Keary	Ernesto Arroyo	Frank Lemmon	Gene McKenna
Hal Lockhart	Izhar By-Gad	Jeremy Poteet	José Pedro Arroyo	K.K. Mookhey
Kevin McLaughlin	Mark Curphey	Martin Eizner	Mikael Simonsson	Neal Krawetz
Nigel Tranter	Raoul Endres	Ray Stirbei	Richard Parke	Robert Hansen
Roy McNamara	Steve Taylor	Sverre Huseby	Tim Smith	William Hau

Part II. Body Matter

Table of Contents

Chapter 1. Chapter 1

About the Open Web Application Security Project

The Open Web Application Security Project (OWASP) is an open community dedicated to finding and fighting the causes of insecure software. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security.

OWASP is a new type of entity in the security market. Our freedom from commercial pressures allows us to provide unbiased, practical, cost-effective information about application security. OWASP is not affiliated with any technology company, although we support the informed use of security technology.

We advocate approaching application security as a people, process, and technology problem. The most effective approaches to application security include improvements in all of these areas.

Structure and Licensing

The OWASP Foundation is the not for profit (501c3) entity that provides the infrastructure for the OWASP community. The Foundation provides our servers and bandwidth, facilitates projects and chapters, and manages the worldwide OWASP AppSec Conferences.

All of the OWASP materials are available under an approved open source license. If you opt to become an OWASP member organization, can also use the commercial license that allows you to use, modify, and distribute all of the OWASP materials within your organization under a single license.

Participation and Membership

Everyone is welcome to participate in our forums, projects, chapters, and conferences. OWASP is a fantastic place to learn about application security, network, and even build your reputation as an expert. Many application security experts and companies participate in OWASP because the community establishes their credibility.

If you get value from the OWASP materials, please consider supporting our cause by becoming an OWASP member. All monies received by the OWASP Foundation go directly into supporting OWASP projects.

Projects

OWASP projects are broadly divided into two main categories, development projects, and documentation projects. Our documentation projects currently consist of:

- The Guide – This document which provides detailed guidance on web application security.
- Top Ten Most Critical Web Application Vulnerabilities – A high level document to help focus on the most critical issues.
- Metrics – A project to define workable web application security metrics.
- Legal – A project to help software buyers and sellers negotiate appropriate security in their contracts.
- Testing Guide – A guide focused on effective web application security testing.
- ISO17799 – Supporting documents for organizations performing ISO17799 reviews.

- AppSec FAQ – Frequently asked questions and answers about application security.

Development projects include:

- WebScarab - a web application vulnerability assessment suite including proxy tools.
- Validation Filters – (Stinger for J2EE, filters for PHP) generic security boundary filters that developers can use in their own applications.
- WebGoat - an interactive training and benchmarking tool that users can learn about web application security in a safe and legal environment.
- DotNet – a variety of tools for securing .NET environments.

Chapter 2. Chapter 2

Introduction

Welcome to the OWASP Guide 2.0!

The Guide has been re-written from the ground up, dealing with all forms of web application security issues, from old hoary chestnuts like SQL injection, through modern concerns such as phishing, credit card handling, session fixation, cross-site request forgeries, and compliance and privacy issues.

In Guide 2.0, you will find details on securing most forms of web applications and services, with practical guidance using J2EE, ASP.NET, and PHP samples. We now use the highly successful OWASP Top 10 style, but with more depth, and references to take you further.

Security is not a black and white field; it is many shades of grey. In the past, many organizations wished to buy a simple silver security bullet – "do it this way or follow this check list to the letter, and you'll be safe." The black and white mindset is invariably wrong, costly, and ineffective.

The Guide strongly recommends the use of threat risk modeling as a way to reduce development costs and time, and eliminate wasted resources. Instead, with careful selection of controls via threat risk modeling, only those controls that demonstrably reduce the risk are implemented. These controls are usually cheap, effective, and simple to implement.

In some countries, risk-based development is not an optional extra, but legally mandated. For example, a core control required by Sarbanes Oxley is to prove that adequate controls are in place for financial systems, and that senior management believes the controls are effective. The Guide provides keys into COBIT (the most commonly used control framework for SOX) to assist organizations produce applications that meet SOX requirements.

As with any long-lived project, there is a need to keep the material fresh and relevant. Therefore, some of the material from the older Guides has been migrated to OWASP's portal or outright replaced with new advice.

Thanks to the many authors and editors for their hard work in bringing this guide to where it is today. If you have any comments or suggestions on the Guide, please e-mail the Guide mail list (see our web site for details) or contact me directly.

Andrew van der Stock,<vanderaj@owasp.org>

Melbourne, Australia

July 24, 2005

Chapter 3. Chapter 3

What are web applications?

Overview

In the early days of the web, web sites consisted of static pages. Obviously, static content prevents the application interacting with the user. As this is limiting, web server manufacturers allowed external programs to run by implementing the Common Gateway Interface (or CGI) mechanism. This allowed input from the user to be sent to an external program or script, processed and then the result rendered back to the user. CGI is the granddaddy of all the various web application frameworks, scripting languages and web services in common use today.

CGI is becoming rare now, but the idea of a process executing dynamic information supplied by the user or a data store, and rendering the dynamic output back is now the mainstay of web applications.

Technologies

CGI

CGI is still used by many sites. An advantage for CGI is the ease of writing the application logic in a fast native language, like C or C++, or to enable a previously non-web enabled application to be accessible via web browsers.

There are several disadvantages to writing applications using CGI:

- Most low level languages do not directly support HTML output, and thus a library needs to be written (or used), or HTML output is manually created on the fly by the programmer
- The write - compile – deploy - run cycle is slower than most of the later technologies (but not hugely so)
- CGI's are a separate process, and the performance penalty of IPC and process creation can be huge on some architectures
- CGI does not support session controls, so a library has to be written or imported to support sessions
- Not everyone is comfortable writing in a low level language (like C or C++), so the barrier of entry is somewhat high, particularly compared to scripting languages
- Most 3rd generation languages commonly used in CGI programs (C or C++) suffer from buffer overflows and resource leaks. This requires a fair amount of skill to avoid.

CGI can be useful in heavy-duty computation or lengthy processes with a smaller number of users.

Filters

Filters are used for specific purposes, such as controlling access to a web site, implementing another web application framework (like Perl, PHP or ASP), or providing a security check. A filter has to be written in C or C++ and can be high performance as it lives within the execution context of the web server itself. Typical examples of a filter interface include Apache web server modules, Sun ONE's NSAPI, and Microsoft's ISAPI. As filters are rarely used specialist interfaces that can directly affect the availability of the web server, they are not considered further.

Scripting

CGI's lack of session management and authorization controls hampered the development of commercially useful web applications. Along with a relatively slower development turn around, web

developers moved to interpreted scripting languages as a solution. The interpreters run script code within the web server process, and as the scripts were not compiled, the write-deploy-run cycle was a bit quicker. Scripting languages rarely suffer from buffer overflows or resource leaks, and thus are easier for programmers to avoid the one of the most common security issues.

There are some disadvantages:

- Most scripting languages aren't strongly typed and do not promote good programming practices
- Scripting languages are generally slower than their compiled counterparts (sometimes as much as 100 times slower)
- Scripts often lead to unmentionable code bases that perform poorly as their size grows
- It's difficult (but not impossible) to write multi-tier large scale applications in scripting languages, so often the presentation, application and data tiers reside on the same machine, limiting scalability and security
- Most scripting languages do not natively support remote method or web service calls, thus making it difficult to communicate with application servers and external web services.

Despite the disadvantages, many large and useful code bases are written in scripting languages, such as eGroupWare (PHP), and many older Internet Banking sites are often written in ASP.

Scripting frameworks include ASP, Perl, Cold Fusion, and PHP. However, many of these would be considered interpreted hybrids now, particularly later versions of PHP and Cold Fusion, which pre-tokenize and optimize scripts.

Web application frameworks

As the boundaries of performance and scalability were being reached by scripting languages, many larger vendors jumped on Sun's J2EE web development platform. J2EE:

- Uses the Java language to produce fast applications (nearly as fast as C++ applications) that do not easily suffer from buffer overflows and memory leaks
- Allowed large distributed applications to run acceptably for the first time
- Possesses good session and authorization controls
- Enabled relatively transparent multi-tier applications via various remote component invocation mechanisms
- Is strongly typed to prevent many common security and programming issues before the program even runs

There are many J2EE implementations available, including the Tomcat reference implementation from the Apache Foundation. The downside is that J2EE has a similar or steeper learning curve to C++, which makes it difficult for web designers and entry-level programmers to write applications. Recently, graphical development tools made it somewhat easier, but compared to PHP, J2EE is still quite a stretch.

Microsoft massively updated their ASP technology to ASP.NET, which uses the .NET Framework and just-in-time MSIL native compilers. .NET Framework in many ways mimicked the J2EE framework, but MS improved on the development process in various ways, such as:

- Easy for entry level programmers and web designers to whip up smaller applications
- Allows large distributed applications

- Possesses good session and authorization controls
- Programmers can use their favorite language, which is compiled to native code for excellent performance (near to C++ speeds), along with buffer overflow and resource garbage collection
- Transparent communication with remote and external components
- is strongly typed to prevent many common security and programming issues before the program even runs

The choice of between J2EE or ASP.NET frameworks is largely dependent upon platform choice. Applications targeting J2EE theoretically can run with few (if any) changes between any of the major vendors and on many platforms from Linux, AIX, MacOS X, or Windows. In practice, some tweaking is required, but complete re-writes are not required.

ASP.Net is primarily available for the Microsoft Windows platform. The Mono project (http://www.go-mono.com/) can run ASP.NET applications on many platforms including Solaris, Netware, and Linux.

There is little reason to choose one over the other from a security perspective.

Common Types of Programming Structures

Small to medium scale applications

Most applications fall into this category. The usual architecture is a simple linear procedural script. This is the most common form of coding for ASP, Cold Fusion and PHP scripts, but rarer (but not impossible) for ASP.NET and J2EE applications.

The reason for this architecture is that it is easy to write, and few skills are required to maintain the code. For smaller applications, any perceived performance benefit from moving to a more scalable architecture will never be recovered in the runtime for those applications. For example, if it takes an additional three weeks of developer time to re-factor the scripts into an MVC approach, the three weeks will never be recovered (or noticed by end users) from the improvements in scalability.

It is typical to find many security issues in such applications, including dynamic database queries constructed from insufficiently validated data input, poor error handling and weak authorization controls.

This Guide provides advice throughout to help improve the security of these applications.

Large scale applications

Larger applications need a different architecture to that of a simple survey or feedback form. As applications get larger, it becomes ever more difficult to implement and maintain features and to keep scalability high. Using scalable application architectures becomes a necessity rather than a luxury when an application needs more than about three database tables or presents more than approximately 20 - 50 functions to a user.

Scalable application architecture is often divided into tiers, and if design patterns are used, often broken down into re-usable chunks using specific guidelines to enforce modularity, interface requirements and object re-use. Breaking the application into tiers allows the application to be distributed to various servers, thus improving the scalability of the application at the expense of complexity.

One of the most common web application architectures is model-view-controller (MVC), which implements the Smalltalk 80 application architecture. MVC is typical of most Apache Foundation Jakarta Struts J2EE applications, and the code-behinds of ASP.NET can be considered a partial implementation of this approach. For PHP, the WACT project (http://wact.sourceforge.net) aims to implement the MVC paradigm in a PHP friendly fashion.

View

The front-end rendering code, often called the presentation tier, should aim to produce the HTML output for the user with little to no application logic.

As many applications will be internationalized (i.e. contain no localized strings or culture information in the presentation layer), they must use calls into the model (application logic) to obtain the data required to render useful information to the user in their preferred language and culture, script direction, and units.

All user input is directed back to controllers in the application logic.

Controller

The controller (or application logic) takes input from the users and gates it through various workflows that call on the application's model objects to retrieve, process, or store the data.

Well written controllers centrally server-side validate input data against common security issues before passing the data to the model for processing, and ensure that output is safe or in a ready form for safe output by the view code.

As the application is likely to be internationalized and accessible, the data needs to be in the local language and culture. For example, dates cannot only be in different orders, but an entirely different calendar could be in use. Applications need to be flexible about presenting and storing data. Simply displaying "9/11/2001" is completely ambiguous to anyone outside a few countries.

Model

Models encapsulate functionality, such as "Account" or "User". A good model should be transparent to the caller, and provide a method to deal with high-level business processes rather than a thin shim to the data store. For example, a good model will allow pseudo code like this to exist in the controller:

```
oAccount->TransferFunds(fromAcct, ToAcct, Amount)
```

rather than writing it like this:

```
if    oAccount->isMyAcct(fromAcct) &
      amount < oAccount->getMaxTransferLimit() &
      oAccount->getBalance(fromAcct) > amount &
      oAccount->ToAccountExists(ToAcct) &
      then
       if oAccount->withdraw(fromAcct, Amount) = OK then
        oAccount->deposit(ToAcct, Amount)
       end if
     end if
```

The idea is to encapsulate the actual dirty work into the model code, rather than exposing primitives. If the controller and model are on different machines, the performance difference will be staggering, so it is important for the model to be useful at a high level.

The model is responsible for checking data against business rules, and any residual risks unique to the data store in use. For example, if a model stores data in a flat file, the code needs to check for OS injection commands if the flat files are named by the user. If the model stores data in an interpreted language, like SQL, then the model is responsible for preventing SQL injection. If it uses a message queue interface to a mainframe, the message queue data format (typically XML) needs to be well formed and compliant with a DTD.

The contract between the controller and the model needs to be carefully considered to ensure that data is strongly typed, with reasonable structure (syntax), and appropriate length, whilst allowing flexibility to allow for internationalization and future needs.

Calls by the model to the data store should be through the most secure method possible. Often the weakest possibility is dynamic queries, where a string is built up from unverified user input. This leads directly to SQL injection and is frowned upon. For more information, see chapter and section .

The best performance and highest security is often obtained through parameterized stored procedures, followed by parameterized queries (also known as prepared statements) with strong typing of the parameters and schema. The major reason for using stored procedures is to minimize network traffic for a multi-stage transaction or to remove security sensitive information from traversing the network.

Stored procedures are not always a good idea – they tie you to a particular database vendor and many implementations are not fast for numeric computation. If you use the 80/20 rule for optimization and move the slow and high-risk transactions to stored procedures, the wins can be worthwhile from a security and performance perspective.

Conclusion

Web applications can be written in many different ways, and in many different languages. Although the Guide concentrates upon three common choices for its examples (PHP, J2EE and ASP.NET), the Guide can be used with any web application technology.

Chapter 4. Chapter 4

Security Architecture and Design

Secure by Design - Policy Frameworks

Overview

Secure applications do not just happen – they are the result of an organization deciding that they will produce secure applications. OWASP's does not wish to force a particular approach or require an organization to pick up compliance with laws that do not affect them - every organization is different.

However, for a secure application, the following at a minimum are required:

- Organizational management which champions security
- Written information security policy properly derived from national standards
- A development methodology with adequate security checkpoints and activities
- Secure release and configuration management

Many of the controls within OWASP Guide 2.0 are influenced by requirements in national standards or control frameworks like COBIT; typically controls selected out of OWASP will satisfy relevant ISO 17799 and COBIT controls.

Organizational Commitment to Security

Organizations that have security buy-in from the highest levels will generally produce and procure applications that meet basic information security principles. This is the first of many steps along the path between ad hoc "possibly secure (but probably not)" to "pretty secure".

Organizations that do not have management buy-in, or simply do not care about security, are extraordinarily unlikely to produce secure applications. Each secure organization documents its "taste" for risk in their information security policy, thus making it easy to determine which risks will be accepted, mitigated, or assigned.

Insecure organizations simply don't know where this "taste" is set, and so when projects run by the insecure organization select controls, they will either end up implementing the wrong controls or not nearly enough controls. Rare examples have been found where every control, including a kitchen sink tealeaf strainer has been implemented, usually at huge cost.

Most organizations produce information security policies derived from ISO 17799, or if in the US, from COBIT, or occasionally both or other standards. There is no hard and fast rule for how to produce information security policies, but in general:

- If you're publicly traded in most countries, you must have an information security policy
- If you're publicly traded in the US, you must have an information security policy which is compliant with SOX requirements, which generally means COBIT controls
- If you're privately held, but have more than a few employees or coders, you probably need one
- Popular FOSS projects, which are not typical organizations, should also have an information security policy

It is perfectly fine to mix and match controls from COBIT and ISO 17799 and almost any other information security standard; rarely do they disagree on the details. The method of production is sometimes tricky – if you "need" certified policy, you will need to engage qualified firms to help you.

OWASP's Place at the Framework table

Organizations need to establish information security policy informed by relevant national legislation, industry regulation, merchant agreements, and subsidiary best practice guides, such as OWASP. As it is impossible to draw a small diagram containing all relevant laws and regulations, you should assume all of the relevant laws, standards, regulations, and guidelines are missing – you need to find out which affect your organization, customers (as applicable), and where the application is deployed.

The following diagram demonstrates where OWASP fits in (substitute your own country and its laws, regulations and standards if it does not appear):

- COBIT
- ISO 17799
- Sarbanes-Oxley

IANAL: OWASP is not a qualified source of legal advice; you should seek your own legal advice.

COBIT

COBIT is a popular risk management framework structured around four domains:

- Planning and organization
- Acquisition and implementation
- Delivery and support
- Monitoring

Each of the four domains has 13 high level objectives, such as DS5 Ensure Systems Security. Each high level objective has a number of detailed objectives, such as 5.2 Identification, Authentication, and Access. Objectives can be fulfilled in a variety of methods that are likely to be different for each organization. COBIT is typically used as a SOX control framework, or as a complement to ISO 17799 controls. OWASP does not dwell on the management and business risk aspects of COBIT. If you are implementing COBIT, OWASP is an excellent start for systems development risks and to ensure that custom and off the shelf applications comply with COBIT controls, but OWASP is not a COBIT compliance magic wand.

Where a COBIT objective is achieved with an OWASP control, you will see "COBIT XXy z.z" to help direct you to the relevant portion of COBIT control documentation. Such controls should be a part of all applications.

For more information about COBIT, please visit:

ISO 17799

ISO 17799 is a risk-based Information Security Management framework directly derived from the AS / NZS 4444 and BS 7799 standards. It is an international standard and used heavily in most organizations not in the US. Although somewhat rare, US organizations use ISO 17799 as well, particularly if they have subsidiaries outside the US. ISO 17799 dates back to the mid-1990's, and some of the control objectives reflect this age – for example calling administrative interfaces "diagnostic ports".

Organizations using ISO 17799 can use OWASP as detailed guidance when selecting and implementing a wide range of ISO 17999 controls, particularly those in the Systems Development chapter, amongst others. Where a 17799 objective is achieved with an OWASP control, you will see "ISO 17799 X.y.z" to help direct you to the relevant portion of ISO 17799. Such controls should be a part of all applications.

For more information about ISO 17799, please visit and the relevant standards bodies, such as Standards Australia ( ), Standards New Zealand ( ), or British Standards International ( ).

Sarbanes-Oxley

A primary motivator for many US organizations in adopting OWASP controls is to assist with ongoing Sarbanes-Oxley compliance. If an organization followed every control in this book, it would not grant the organization SOX compliance. The Guide can be used as a suitable control for application procurement and in-house development, as part of a wider compliance program.

However, SOX compliance is often used as necessary cover for previously resource starved IT managers to implement long neglected security controls, so it is important to understand what SOX actually requires. A summary of SOX, section 404 obtained from AICPA's web site at states:

Section 404: Management Assessment of Internal Controls

Requires each annual report of an issuer to contain an "internal control report", which shall:

1. state the responsibility of management for establishing and maintaining an adequate internal control structure and procedures for financial reporting; and

2. contain an assessment, as of the end of the issuer's fiscal year, of the effectiveness of the internal control structure and procedures of the issuer for financial reporting.

This essentially states that management must establish and maintain internal control structures and procedures, and an annual evaluation that the controls are effective. As finance is no longer conducted using double entry in ledger books, "SOX compliance" is often extended to mean IT.

The Guide can assist SOX compliance by providing effective controls for all applications, and not just for the purposes of financial reporting. It allows organizations to buy products which claim they use OWASP controls, or allow organizations to dictate to custom software houses that they must use OWASP controls to produce more secure software.

However, SOX should not be used as an excuse. SOX controls are necessary to prevent another Enron, not to buy widgets that may or may not help. All controls, whether off the shelf widgets, training, code controls, or process changes, should be selected based on measurable efficacy and ability to treat risk, and not "tick the boxes".

Development Methodology

High performing development shops have chosen a development methodology and coding standards. The choice of development methodology is not as important as simply having one.

Ad hoc development is not structured enough to produce secure applications. Organizations who wish to produce secure code all the time need to have a methodology that supports that goal. Choose the right methodology – small teams should never consider heavy weight methodologies that identify many different roles. Large teams should choose methodologies that scale to their needs.

Attributes to look for in a development methodology:

- Strong acceptance of design, testing and documentation

- Places where security controls (such as threat risk analysis, peer reviews, code reviews, etc) can be slotted in

- Works for the organization's size and maturity

- Has the potential to reduce the current error rate and improve developer productivity

Coding Standards

Methodologies are not coding standards; each shop will need to determine what to use based upon either common practice, or simply to lay down the law based upon known best practices.

Artifacts to consider:

- Architectural guidance (i.e., “web tier is not to call the database directly”)
- Minimum levels of documentation required
- Mandatory testing requirements
- Minimum levels of in-code comments and preferred comment style
- Use of exception handling
- Use of flow of control blocks (e.g., “All uses of conditional flow are to use explicit statement blocks”)
- Preferred variable, function, class and table method naming
- Prefer maintainable and readable code over clever or complex code

Indent style and tabbing are a holy war, and from a security perspective, they simply do not matter that much. However, it should be noted that we no longer use 80x24 terminals, so vertical space usage is not as important as it once was. Indent and tabbing can be “fixed” using automated tools or simply a style within a code editor, so do not get overly fussy on this issue.

Source Code Control

High performing software engineering requires the use of regular improvements to code, along with associated testing regimes. All code and tests must be able to be reverted and versioned.

This could be done by copying folders on a file server, but it is better performed by source code revision tools, such as Git, Subversion, CVS, SourceSafe, or ClearCase.

Why include tests in a revision? Tests for later builds do not match the tests required for earlier builds. It is vital that a test is applied to the build for which it was built.

Chapter 5. Chapter 5

Secure Coding Principles

Overview

Architects and solution providers need guidance to produce secure applications by design, and they can do this by not only implementing the basic controls documented in the main text, but also referring back to the underlying “Why?” in these principles. Security principles such as confidentiality, integrity, and availability – although important, broad, and vague – do not change. Your application will be the more robust the more you apply them.

For example, it is a fine thing when implementing data validation to include a centralized validation routine for all form input. However, it is a far finer thing to see validation at each tier for all user input, coupled with appropriate error handling and robust access control.

In the last year or so, there has been a significant push to standardize terminology and taxonomy. This version of the Guide has normalized its principles with those from major industry texts, while dropping a principle or two present in the first edition of the Guide. This is to prevent confusion and to increase compliance with a core set of principles. The principles that have been removed are adequately covered by controls within the text.

Asset Classification

Selection of controls is only possible after classifying the data to be protected. For example, controls applicable to low value systems such as blogs and forums are different to the level and number of controls suitable for accounting, high value banking and electronic trading systems.

About attackers

When designing controls to prevent misuse of your application, you must consider the most likely attackers (in order of likelihood and actualized loss from most to least):

- Disgruntled staff or developers
- “Drive by” attacks, such as side effects or direct consequences of a virus, worm or Trojan attack
- Motivated criminal attackers, such as organized crime
- Criminal attackers without motive against your organization, such as defacers
- Script kiddies

Notice there is no entry for the term “hacker.” This is due to the emotive and incorrect use of the word “hacker” by the media. The correct term is “criminal.” The typical criminal caught and prosecuted by the police are script kiddies, mainly due to organizations being unwilling to go to the police and help them lay charges against the more serious offenders.

However, it is far too late to reclaim the incorrect use of the word “hacker” and try to return the word to its correct roots. The Guide consistently uses the word “attacker” when denoting something or someone who is actively attempting to exploit a particular feature.

Core pillars of information security

Information security has relied upon the following pillars:

- Confidentiality – only allow access to data for which the user is permitted
- Integrity – ensure data is not tampered or altered by unauthorized users
- Availability – ensure systems and data are available to authorized users when they need it

The following principles are all related to these three pillars. Indeed, when considering how to construct a control, considering each pillar in turn will assist in producing a robust security control.

Security Architecture

Applications without security architecture are as bridges constructed without finite element analysis and wind tunnel testing. Sure, they look like bridges, but they will fall down at the first flutter of a butterfly's wings. The need for application security in the form of security architecture is every bit as great as in building or bridge construction.

Application architects are responsible for constructing their design to adequately cover risks from both typical usage, and from extreme attack. Bridge designers need to cope with a certain amount of cars and foot traffic but also cyclonic winds, earthquake, fire, traffic incidents, and flooding. Application designers must cope with extreme events, such as brute force or injection attacks, and fraud. The risks for application designers are well known. The days of "we didn't know" are long gone. Security is now expected, not an expensive add-on or simply left out.

Security architecture refers to the fundamental pillars: the application must provide controls to protect the confidentiality of information, integrity of data, and provide access to the data when it is required (availability) – and only to the right users. Security architecture is not "markitecture", where a cornucopia of security products are tossed together and called a "solution", but a carefully considered set of features, controls, safer processes, and default security posture.

When starting a new application or re-factoring an existing application, you should consider each functional feature, and consider:

- Is the process surrounding this feature as safe as possible? In other words, is this a flawed process?
- If I were evil, how would I abuse this feature?
- Is the feature required to be on by default? If so, are there limits or options that could help reduce the risk from this feature?

Andrew van der Stock calls the above process , and recommends putting yourself in the shoes of the attacker and thinking through all the possible ways you can abuse each and every feature, by considering the three core pillars and using the STRIDE model in turn.

By following this guide, and using the STRIDE / DREAD threat risk modeling discussed here and in Howard and LeBlanc's book, you will be well on your way to formally adopting a security architecture for your applications.

The best system architecture designs and detailed design documents contain security discussion in each and every feature, how the risks are going to be mitigated, and what was actually done during coding.

Security architecture starts on the day the business requirements are modeled, and never finish until the last copy of your application is decommissioned. Security is a life-long process, not a one shot accident.

Security Principles

These security principles have been taken from the previous edition of the OWASP Guide and normalized with the security principles outlined in Howard and LeBlanc's excellent Writing Secure Code.

Minimize Attack Surface Area

Every feature that is added to an application adds a certain amount of risk to the overall application. The aim for secure development is to reduce the overall risk by reducing the attack surface area.

For example, a web application implements online help with a search function. The search function may be vulnerable to SQL injection attacks. If the help feature was limited to authorized users, the attack likelihood is reduced. If the help feature's search function was gated through centralized data validation routines, the ability to perform SQL injection is dramatically reduced. However, if the help feature was re-written to eliminate the search function (through better user interface, for example), this almost eliminates the attack surface area, even if the help feature was available to the Internet at large.

Secure Defaults

There are many ways to deliver an "out of the box" experience for users. However, by default, the experience should be secure, and it should be up to the user to reduce their security – if they are allowed.

For example, by default, password aging and complexity should be enabled. Users might be allowed to turn these two features off to simplify their use of the application and increase their risk.

Principle of Least Privilege

The principle of least privilege recommends that accounts have the least amount of privilege required to perform their business processes. This encompasses user rights, resource permissions such as CPU limits, memory, network, and file system permissions.

For example, if a middleware server only requires access to the network, read access to a database table, and the ability to write to a log, this describes all the permissions that should be granted. Under no circumstances should the middleware be granted administrative privileges.

Principle of Defense in Depth

The principle of defense in depth suggests that where one control would be reasonable, more controls that approach risks in different fashions are better. Controls, when used in depth, can make severe vulnerabilities extraordinarily difficult to exploit and thus unlikely to occur.

With secure coding, this may take the form of tier-based validation, centralized auditing controls, and requiring users to be logged on all pages.

For example, a flawed administrative interface is unlikely to be vulnerable to anonymous attack if it correctly gates access to production management networks, checks for administrative user authorization, and logs all access.

Fail securely

Applications regularly fail to process transactions for many reasons. How they fail can determine if an application is secure or not.

For example:

```
isAdmin = true;
    try {
    codeWhichMayFail();
    isAdmin = isUserInRole( “Administrator” );
    }
    catch (Exception ex) {
    log.write(ex.toString());
     }
```

If codeWhichMayFail() fails, the user is an admin by default. This is obviously a security risk.

External Systems are Insecure

Many organizations utilize the processing capabilities of third party partners, who more than likely have differing security policies and posture than you. It is unlikely that you can influence or control any external third party, whether they are home users or major suppliers or partners.

Therefore, implicit trust of externally run systems is not warranted. All external systems should be treated in a similar fashion.

For example, a loyalty program provider provides data that is used by Internet Banking, providing the number of reward points and a small list of potential redemption items. However, the data should be checked to ensure that it is safe to display to end users, and that the reward points are a positive number, and not improbably large.

Separation of Duties

A key fraud control is separation of duties. For example, someone who requests a computer cannot also sign for it, nor should they directly receive the computer. This prevents the user from requesting many computers, and claiming they never arrived.

Certain roles have different levels of trust than normal users. In particular, Administrators are different to normal users. In general, administrators should not be users of the application.

For example, an administrator should be able to turn the system on or off, set password policy but shouldn't be able to log on to the storefront as a super privileged user, such as being able to "buy" goods on behalf of other users.

Do not trust Security through Obscurity

Security through obscurity is a weak security control, and nearly always fails when it is the only control. This is not to say that keeping secrets is a bad idea, it simply means that the security of key systems should not be reliant upon keeping details hidden.

For example, the security of an application should not rely upon knowledge of the source code being kept secret. The security should rely upon many other factors, including reasonable password policies, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls.

A practical example is Linux. Linux's source code is widely available, and yet when properly secured, Linux is a hardy, secure and robust operating system.

Simplicity

Attack surface area and simplicity go hand in hand. Certain software engineering fads prefer overly complex approaches to what would otherwise be relatively straightforward and simple code.

Developers should avoid the use of double negatives and complex architectures when a more simple approach would be faster and easier to read.

For example, although it might be fashionable to have a slew of singleton entity beans running on a separate middleware server, it is more secure and faster to simply use global variables with an appropriate mutex mechanism to protect against race conditions.

Fix Security Issues Correctly

Once a security issue has been identified, it is important to develop a test for it, and to understand the root cause of the issue. When design patterns are used, it is likely that the security issue is widespread amongst all code bases, so developing the right fix without introducing regressions is essential.

For example, a user has found that they can see another user's balance by adjusting their cookie. The fix seems to be relatively straightforward, but as the cookie handling code is shared amongst all applications, a change to just one application will trickle through to all other applications. The fix must therefore be tested on all affected applications.

Part III. End Matter

Table of Contents

Chapter 6. Chapter Template Title

Text