

# Paranoia-101

They are out to get you ...

Your web site is insecure ...

## Develop a healthy paranoia

- The hackers are out to get you.
- So, put some defences into your web site.

## Topics

1. Common Weakness Enumeration
  - From SANS (SysAdmin, Audit, Network, Security) Institute and Mitre Corporation
  - Identifies most common weaknesses in applications and provides some suggestions as to how to ameliorate them
2. An example of a hacker attack via SQL injection
3. Threat Modelling
  - An approach to analysing an application to identify, evaluate, and mitigate security risks.

## CWE

Common Weakness Enumeration

## Common Weakness Enumeration

- <http://www.sans.org/top25errors/>
- <http://cwe.mitre.org>
- Exploring common problems and ranking these according to damage caused
- Recently published the "Top 25" – identifying three main classes
  - Incorrect Interaction Between Components
  - Risky Resource Management
  - Porous Defences

CWE is a Software Assurance strategic initiative sponsored by the National Cyber Security Division of the U.S. Department of Homeland Security.

## CAPEC

- **Common Attack Pattern Enumeration and Classification (CAPEC)**
- <http://capec.mitre.org/>
- Models of how attacks can be made on weaknesses
  - CWE and CAPEC entries interlinked

CAPEC is sponsored by the National Cyber Security Division of the U.S. Department of Homeland Security.

## CWE

- CWE analysing all programs
- But many of examples are clearly "web related"
- Most attacks likely to be external sourced (well, I suppose that depends on the kind of employee you've hired)
- External attacks come through Internet
- Web is core to so many applications
  - Which is why it's left to a 300-level elective subject.

## Problems

- Accountability
- Authentication
  - Attack may subvert authentication checks – allowing for imposters to gain access
- Authorization
  - Attack may increase permissions allowed to request
- Availability
  - Attack may make service/resource unavailable
- Confidentiality
- Integrity
  - Inappropriate modification of data values

## Insecure Interaction Between Components

- These weaknesses are related to insecure ways in which data is sent and received between separate components, modules, programs, processes, threads, or systems.
  - CWE-20: Improper Input Validation
  - CWE-116: Improper Encoding or Escaping of Output
  - CWE-89: Failure to Preserve SQL Query Structure (aka 'SQL Injection')
  - CWE-79: Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')
  - CWE-78: Failure to Preserve OS Command Structure (aka 'OS Command Injection')
  - CWE-319: Cleartext Transmission of Sensitive Information
  - CWE-352: Cross-Site Request Forgery (CSRF)
  - CWE-362: Race Condition
  - CWE-209: Error Message Information Leak

## Risky Resource Management

- The weaknesses in this category are related to ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources.
  - CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer
  - CWE-642: External Control of Critical State Data
  - CWE-73: External Control of File Name or Path
  - CWE-426: Untrusted Search Path
  - CWE-94: Failure to Control Generation of Code (aka 'Code Injection')
  - CWE-494: Download of Code Without Integrity Check
  - CWE-404: Improper Resource Shutdown or Release
  - CWE-665: Improper Initialization
  - CWE-682: Incorrect Calculation

## Porous Defences

- The weaknesses in this category are related to defensive techniques that are often misused, abused, or just plain ignored.
  - CWE-285: Improper Access Control (Authorization)
  - CWE-327: Use of a Broken or Risky Cryptographic Algorithm
  - CWE-259: Hard-Coded Password
  - CWE-732: Insecure Permission Assignment for Critical Resource
  - CWE-330: Use of Insufficiently Random Values
  - CWE-250: Execution with Unnecessary Privileges
  - CWE-602: Client-Side Enforcement of Server-Side Security



**Insecure Interaction**

### CWE-20: Improper Input Validation

- Summary
  - The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program.
- Extended Description
  - When software fails to validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application. This will lead to parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution.

**Insecure Interaction**

### CWE-20: Consequences

- Availability
  - An attacker could provide unexpected values and cause a program crash.
- Confidentiality
  - An attacker could read confidential data if they are able to control resource references.
- Integrity
  - An attacker could modify data or possibly alter control flow in unexpected ways.

**Insecure Interaction**

### CWE-20: Advice

- Architecture/design stage
  - Use a standard input validation framework
  - Identify all inputs: parameters, request headers, URLs, cookies, environment variables – Validate them all.
  - Assume that input is malicious; use things like "whitelist" (e.g. a "Realtor web site" should check that a requested 'suburb' is a member of a set of all known suburbs)
  - Repeat any checks done in client
    - (If find data that should have been rejected by client side checks, might be worth logging as evidence of hacker probes)

**Insecure Interaction**

### CWE-20: Advice

- Implementation stage
  - If combining data from multiple sources, check after combination as well as checking individual data items.
  - Check strings representing numeric values by using standard conversion functions to get value and then check value
  - Ensure common character encoding among components
  - ...

**Insecure Interaction**

### CWE-89: Failure to Preserve SQL Query Structure (aka 'SQL Injection')

- Summary
  - The application dynamically generates an SQL query based on user input, but it does not sufficiently prevent that input from modifying the intended structure of the query.
- Extended Description
  - Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. This can be used to alter query logic to bypass security checks, or to insert additional statements that modify the back-end database, possibly including execution of system commands.
  - SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

Insecure Interaction

## CWE-89: Consequences

- Confidentiality
  - Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.
- Authentication
  - If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- Authorization
  - If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL injection vulnerability.
- Integrity
  - Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack.

Insecure Interaction

## CWE-89: Advice

- Architecture and Design stage
  - Process SQL queries using prepared statements, parameterized queries, or stored procedures. These features should accept parameters or variables and support strong typing. Do not dynamically construct and execute query strings within these features using "exec" or similar functionality, since you may re-introduce the possibility of SQL injection.
  - ...

Later – example illustrating "white hat hacker" use of SQL injection to probe weaknesses in a web site

Insecure Interaction

## CWE-79: Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')

- Summary
  - The software does not sufficiently validate, filter, escape, and encode user-controllable input before it is placed in output that is used as a web page that is served to other users.
- Extended Description
  - Cross-site scripting (XSS) vulnerabilities occur when:
    1. Untrusted data enters a web application, typically from a web request.
    2. The web application dynamically generates a web page that contains this untrusted data.
    3. During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX, etc.
    4. A victim visits the generated web page through a web browser, which contains malicious script that was injected using the untrusted data.
    5. Since the script comes from a web page that was sent by the web server, the web browser executes the malicious script in the context of the web server's domain.
    6. This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.

Insecure Interaction

## CWE-79: XSS

- Type 1: Reflected XSS (or Non-Persistent)
  - The server reads data directly from the HTTP request and reflects it back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser.

The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

Insecure Interaction

## CWE-79: XSS

- Type 2: Stored XSS (or Persistent)
  - The application stores dangerous data in a database, message forum, visitor log, or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content.

From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

Insecure Interaction

## CWE-79: XSS

- Type 0: DOM-Based XSS
  - In DOM-based XSS, the client performs the injection of XSS into the page; in the other types, the server performs the injection. DOM-based XSS generally involves server-controlled, trusted script that is sent to the client, such as Javascript that performs sanity checks on a form before the user submits it. If the server-supplied script processes user-supplied data and then injects it back into the web page (such as with dynamic HTML), then DOM-based XSS is possible.

Insecure Interaction

## CWE-89: Consequences

- Confidentiality
  - The most common attack performed with cross-site scripting involves the disclosure of information stored in user cookies. Typically, a malicious user will craft a client-side script, which -- when parsed by a web browser -- performs some activity (such as sending all site cookies to a given E-mail address). This script will be loaded and run by each user visiting the web site. Since the site requesting to run the script has access to the cookies in question, the malicious script does also.
- Access Control
  - In some circumstances it may be possible to run arbitrary code on a victim's computer when cross-site scripting is combined with other flaws.
- Confidentiality, Integrity, Availability
  - The consequence of an XSS attack is the same regardless of whether it is stored or reflected. The difference is in how the payload arrives at the server.
  - XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. Some cross-site scripting vulnerabilities can be exploited to manipulate or steal cookies, create requests that can be mistaken for those of a valid user, compromise confidential information, or execute malicious code on the end user systems for a variety of nefarious purposes. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, running "Active X" controls (under Microsoft Internet Explorer) from sites that a user perceives as trustworthy, and modifying presentation of content.

Insecure Interaction

## CWE-89: Advice

- Various including *RSnake*. "*XSS (Cross Site Scripting) Cheat Sheet*".  
<<http://ha.ckers.org/xss.html>>.  
This is guide to testing a site for XSS vulnerabilities

Insecure Interaction

## CWE-209: Error Message Information Leak

- Summary
  - The software generates an error message that includes sensitive information about its environment, users, or associated data.
- Extended Description
  - The sensitive information may be valuable information on its own (such as a password), or it may be useful for launching other, more deadly attacks. If an attack fails, an attacker may use error information provided by the server to launch another more focused attack. For example, an attempt to exploit a path traversal weakness (CWE-22) might yield the full pathname of the installed application. In turn, this could be used to select the proper number of "." sequences to navigate to the targeted file. An attack using SQL injection (CWE-89) might not initially succeed, but an error message could reveal the malformed query, which would expose query logic and possibly even passwords or other sensitive information used within the query.

Insecure Interaction

## CWE-209: Advice

- Implementation
  - Ensure that error messages only contain minimal information that are useful to their intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can help an attacker craft another attack that now will pass through the validation filters. If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.
  - Handle exceptions internally and do not display errors containing potentially sensitive information to a user.
- Build and Compilation
  - Debugging information should not make its way into a production release.

Risky Resource Management

## CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer

- Summary
  - The software may potentially allow operations, such as reading or writing, to be performed at addresses not intended by the developer.
- Extended Description
  - When software permits read or write operations on memory located outside of an allocated range, an attacker may be able to access/modify sensitive information, cause the system to crash, alter the intended control flow, or execute arbitrary code.

Risky Resource Management

## CWE-119: Consequences

- Integrity
  - If the memory accessible by the attacker can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow.  
If the attacker can overwrite a pointer's worth of memory (usually 32 or 64 bits), he can redirect a function pointer to his own malicious code. Even when the attacker can only modify a single byte arbitrary code execution can be possible. Sometimes this is because the same problem can be exploited repeatedly to the same effect. Other times it is because the attacker can overwrite security-critical application-specific data -- such as a flag indicating whether the user is an administrator.
- Availability
  - Out of bounds memory access will very likely result in the corruption of relevant memory, and perhaps instructions, possibly leading to a crash. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop.

Risky Resource Management

## CWE-119: Advice

- Requirements
  - Use a language with features that can automatically mitigate or eliminate buffer overflows.
  - For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows.
- Implementation
  - Double check that your buffer is as large as you specify.
  - When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string.
  - Check buffer boundaries if calling this function in a loop and make sure you are not in danger of writing past the allocated space.
  - If necessary, truncate all input strings to a reasonable length before passing them to the copy and concatenation functions.

Risky Resource Management

## CWE-119: Advice

- Operation
  - Use a feature like Address Space Layout Randomization (ASLR). This is not a complete solution. However, it forces the attacker to guess an unknown value that changes every program execution.
  - Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent. This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required

Risky Resource Management

## CWE-119: Links to CAPEC

- CWE entries mostly contain links to CAPEC attack patterns, e.g. CWE-119 references
  - 8Buffer Overflow in an API Call
  - 9Buffer Overflow in Local Command-Line Utilities
  - 10Buffer Overflow via Environment Variables
  - 14Client-side Injection-induced Buffer Overflow
  - 24Filter Failure through Buffer Overflow
  - 42MIME Conversion
  - 44Overflow Binary Resource File
  - 45Buffer Overflow via Symbolic Links
  - 100Overflow Buffers
  - 46Overflow Variables and Tags
  - 47Buffer Overflow via Parameter Expansion
- Each CAPEC entry has a little more technical discussion

Risky Resource Management

## CWE-426: Untrusted Search Path

- Summary
  - The application searches for critical resources using an externally-supplied search path that can point to resources that are not under the application's direct control.
- Extended Description
  - This might allow attackers to execute their own programs, access unauthorized data files, or modify configuration in unexpected ways. If the application uses a search path to locate critical resources such as programs, then an attacker could modify that search path to point to a malicious program, which the targeted application would then execute. The problem extends to any type of critical resource that the application trusts.

Risky Resource Management

## CWE-426: Consequences

- Integrity
  - There is the potential for arbitrary code execution with privileges of the vulnerable program.
- Availability
  - The program could be redirected to the wrong files, potentially triggering a crash or hang when the targeted file is too large or does not have the expected format.
- Confidentiality
  - The program could send the output of unauthorized files to the attacker.

Risky Resource Management

## CWE-426: Example

- **CWE-426 attack is an internal hacker** (or an external hacker who is now able to log on to an internal account).
- CWE notes show how trusted program can be subverted if attacker can change the PATH (or LD\_LIBRARY\_PATH) variable used by that program
  - Attacker places malicious substitute versions in some working directory
  - PATH changed to get that scanned before default libraries and command directories
  - Trusted program runs command (or links to library) but gets malicious code rather than desired code

## CWE-426: Advice

- Implementation
  - When invoking other programs, specify those programs using fully-qualified pathnames.
  - Sanitize your environment before invoking other programs. This includes the PATH environment variable, LD\_LIBRARY\_PATH and other settings that identify the location of code libraries, and any application-specific search paths.
  - Check your search path before use and remove any elements that are likely to be unsafe, such as the current working directory or a temporary files directory.
  - Use other functions that require explicit paths.
    - system() in C does not require a full path - unsafe
    - execl() and execv() require a full path - safer

## CWE-665: Improper Initialization

- Summary
  - The software does not follow the proper procedures for initializing a resource, which might leave the resource in an improper state when it is accessed or used.
- Extended Description
  - This can have security implications when the associated resource is expected to have certain properties or values, such as a variable that determines whether a user has been authenticated or not.

## CWE-665: Improper Initialization

- Summary
  - The software does not follow the proper procedures for initializing a resource, which might leave the resource in an improper state when it is accessed or used.
- Extended Description
  - This can have security implications when the associated resource is expected to have certain properties or values, such as a variable that determines whether a user has been authenticated or not.

## CWE-665: Consequences

- Confidentiality
  - When reusing a resource such as memory or a program variable, the original contents of that resource may not be cleared before it is sent to an untrusted party.
- Integrity
  - The uninitialized data may contain values that cause program flow to change in ways that the programmer did not intend. For example, if an uninitialized variable is used as an array index in C, then its previous contents may produce an index that is outside the range of the array. Alternately, if security-critical decisions rely on a variable having a "0" or equivalent value, and the programming language performs this initialization on behalf of the programmer, then a bypass of security may occur.
- Availability
  - The resource may have values that the program did not expect, causing erroneous code paths to be exercised and leading to a crash or exit.

## CWE-665: Advice

- Requirements
  - Use a language with features that can automatically mitigate or eliminate weaknesses related to initialization. For example, in Java, if the programmer does not explicitly initialize a variable, then the code could produce a compile-time error (if the variable is local) or automatically initialize the variable to the default value for the variable's type. In Perl, if explicit initialization is not performed, then a default value of undef is assigned, which is interpreted as 0, false, or an equivalent value depending on the context in which the variable is accessed.
- Architecture and Design
  - Identify all variables and data stores that receive information from external sources, and apply input validation to make sure that they are only initialized to expected values.
- ...

## CWE-665: Advice

- Implementation
  - Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage.
  - Implementation
    - Pay close attention to complex conditionals that affect initialization, since some conditions might not perform the initialization.
    - Avoid race conditions (CWE-362) during initialization routines.
    - Build and Compilation
      - Run or compile your software with settings that generate warnings about uninitialized variables or data.
- Testing
  - Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.

Porous Defence

## CWE-327: Use of a Broken or Risky Cryptographic Algorithm

- Summary
  - The use of a broken or risky cryptographic algorithm is an unnecessary risk that may result in the disclosure of sensitive information.
- Extended Description
  - The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Well-known techniques may exist to break the algorithm.

Porous Defence

## CWE-327: Consequences

- Confidentiality
  - The confidentiality of sensitive data may be compromised by the use of a broken or risky cryptographic algorithm.
- Integrity
  - The integrity of sensitive data may be compromised by the use of a broken or risky cryptographic algorithm.
- Accountability
  - Any accountability to message content preserved by cryptography may be subject to attack.

Porous Defence

## CWE-327: Advice

- Architecture and Design
  - Do not develop your own cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If your algorithm can be compromised if attackers find out how it works, then it is especially weak.
  - Use a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations. Periodically ensure that you aren't using obsolete cryptography. Some older algorithms, once thought to require a billion years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms which were once regarded as strong.
  - Design your software so that you can replace one cryptographic algorithm with another. This will make it easier to upgrade to stronger algorithms.
  - Carefully manage and protect cryptographic keys (see CWE-320). If the keys can be guessed or stolen, then the strength of the cryptography itself is irrelevant.
- ...

Porous Defence

## CWE-327: Advice

- Implementation
  - Use languages, libraries, or frameworks that make it easier to use strong cryptography. Industry-standard implementations will save you development time and may be more likely to avoid errors that can occur during implementation of cryptographic algorithms.
  - When you use industry-approved techniques, you need to use them correctly. Don't cut corners by skipping resource-intensive steps. These steps are often essential for preventing common attacks.
- Testing
  - Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

Porous Defence

## CWE-259: Hard-Coded Password

- *"Hard-coding a secret account and password into your software's authentication module is extremely convenient - for skilled reverse engineers."*
- Summary
  - The software contains a hard-coded password, which it uses for its own inbound authentication or for outbound communication to external components.

(Example – real – from late 1990s: large DEC computers came with a pre-initialized account that allowed DEC engineers to log in and remotely run diagnostics if the hardware wasn't working fully. The account had a fixed password – that got to be known to persons other than the DEC engineering support staff.)

Porous Defence

## CWE-259: Hard-Coded Password

- Extended Description
  - A hard-coded password typically leads to a significant authentication failure that can be difficult for the system administrator to detect. Once detected, it can be difficult to fix, so the administrator may be forced into disabling the product entirely. There are two main variations:
    - Inbound: the software contains an authentication mechanism that checks for a hard-coded password.
    - Outbound: the software connects to another system or component, and it contains hard-coded password for connecting to that component.
  - In the Inbound variant, a default administration account is created, and a simple password is hard-coded into the product and associated with that account. This hard-coded password is the same for each installation of the product, and it usually cannot be changed or disabled by system administrators without manually modifying the program, or otherwise patching the software. If the password is ever discovered or published (a common occurrence on the Internet), then anybody with knowledge of this password can access the product. Finally, since all installations of the software will have the same password, even across different organizations, this enables massive attacks such as worms to take place.
  - The Outbound variant applies to front-end systems that authenticate with a back-end service. The back-end service may require a fixed password which can be easily discovered. The programmer may simply hard-code those back-end credentials into the front-end software. Any user of that program may be able to extract the password. Client-side systems with hard-coded passwords pose even more of a threat, since the extraction of a password from a binary is usually very simple.

## CWE-259: Likely scenario

- Every "two-tier" application
  - The client end almost certainly has a password for a database
  - Client code distributed
  - Client code reverse engineered
    - E.g. "javap" applied to an applet or Java client application
  - Hacker has database password

## CWE-259: Advice

- Architecture and Design
  - For inbound authentication: apply strong one-way hashes to your passwords and store those hashes in a configuration file or database with appropriate access control. That way, theft of the file/database still requires the attacker to try to crack the password. When handling an incoming password during authentication, take the hash of the password and compare it to the hash that you have saved. Use randomly assigned salts for each separate hash that you generate. This increases the amount of computation that an attacker needs to conduct a brute-force attack, possibly limiting the effectiveness of the rainbow table method.

## CWE-732: Insecure Permission Assignment for Critical Resource

- Summary
  - The software specifies permissions for a security-critical resource in a way that allows that resource to be read or modified by unintended actors.
- Extended Description
  - When a resource is given a permissions setting that provides access to a wider range of actors than required, it could lead to the disclosure of sensitive information, or the modification of that resource by unintended parties. This is especially dangerous when the resource is related to program configuration, execution or sensitive user data.

## CWE-732: Advice

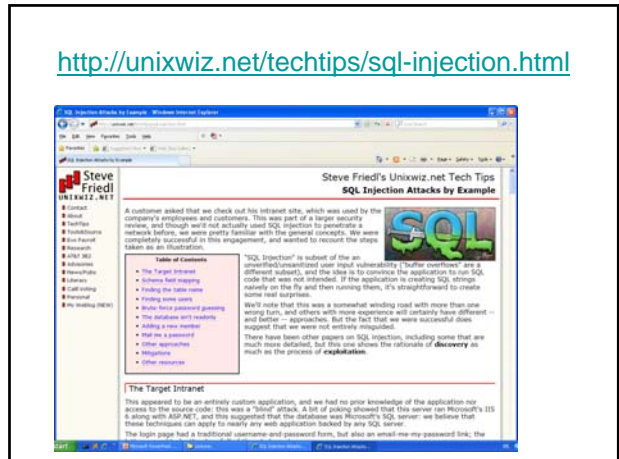
- Architecture and Design
  - When using a critical resource such as a configuration file, check to see if the resource has insecure permissions (such as being modifiable by any regular user), and generate an error or even exit the software if there is a possibility that the resource could have been modified by an unauthorized party.
  - Divide your application into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully defining distinct user groups, privileges, and/or roles. Map these against data, functionality, and the related resources. Then set the permissions accordingly. This will allow you to maintain more fine-grained control over your resources.
- Implementation
  - During program startup, explicitly set the default permissions to the most restrictive setting possible. Also set the appropriate permissions during program installation. This will prevent you from inheriting insecure permissions from any user who installs or runs the program.

## CWE-732: Advice

- System Configuration
  - For all configuration files, executables, and libraries, make sure that they are only readable and writable by the software's administrator.
- Documentation
  - Do not suggest insecure configuration changes in your documentation, especially if those configurations can extend to resources and other software that are outside the scope of your own software.
- Installation
  - Do not assume that the system administrator will manually change the configuration to the settings that you recommend in the manual.

## CWE

- Code that you have written will have some of the weaknesses listed at CWE.
- Skim read the documentation
- Try to recognize those cases where you have made the kinds of errors described.
- Have a look at the CAPEC attack patterns as well.



### Context

- A bit of poking showed that this server ran Microsoft's IIS 6 along with ASP.NET, and this suggested that the database was Microsoft's SQL server.
- The login page had a traditional username-and-password form, but also an email-me-my-password link; the latter proved to be the downfall of the whole system.
- When entering an email address, the system presumably looked in the user database for that email address, and mailed something to that address.

### First test – see if they have CWE-89

- The first test in any SQL-ish form is to enter a single quote as part of the data: the intention is to see if they construct an SQL string literally without sanitizing.
- When submitting the form with a quote in the email address, we get a 500 error (server failure), and this suggests that the "broken" input is actually being parsed literally.
- Bingo.
  - **CWE-89: Failure to Preserve SQL Query Structure**

### Can guess the code style ...

- Guess

```
SELECT fieldlist FROM table WHERE field = '$EMAIL';
```

- Enter **steve@unixwiz.net'** - note the closing quote mark - this yields constructed SQL:

```
SELECT fieldlist ... WHERE field = 'steve@unixwiz.net'';
```

- Repeated single quote character – syntax error, not trapped in code, hence server side died and web server returns the 500 error code

### OK, try adding carefully crafted inputs ...

- Change the nature of that clause *in an SQL legal way* and see what happens. By entering **anything' OR 'x'='x**, the resulting SQL is:

```
SELECT fieldlist FROM table WHERE 'anything' OR 'x'='x';
```

- Program will now get a resultset with many rows (all rows in table!) rather than a single row.
- What will happen?
- Who knows! It depends on how program was written – but most likely the code assumes that one row will be returned, and it will simply work with data in the first row of the multirow resultset

*This doesn't work with all databases; Oracle's SQL parser appears to be fussier and will reject many of the carefully hacked up input strings.*

## What did happen?

- Response page said  
*Your login information has been mailed to random.person@example.com*
  - (random.person was actually the email entered; and example.com was really the company name)
- So, it's a typical poorly written fragment of code.
- But how do you squeeze anything more useful out of it?

## Start guessing names of columns in data table!

- Guess column name for email – probably "email"
- Try an input like:  
`x' AND email IS NULL; --'`
- If we get a server error, it means our SQL is malformed and a syntax error was thrown: it's most likely due to a bad field name. If we get any kind of valid response, we guessed the name correctly.
  - don't care about matching the email address (which is why we use a dummy 'x'),
  - `--` marks the start of an SQL comment; this hides the final quote that the application code would have added (which if present would have always resulted in an SQL syntax error)

## They guessed ...

- Found several valid field names:
  - email
  - passwd
  - login\_id
  - full\_name
- *(It also suggests that programmer is probably using the database column names as the names of fields in the various input forms – so have a look at all the forms associated with the site and pick up more likely field names.)*
- Next, try to find name of table

## Guessing table names ...

- The application's built-in query already has the table name built into it, but we don't know what that name is: there are several approaches for finding that (and other) table names.
- These hackers relied on a **subselect**.
- They have already found a way of getting their SQL into the queries that are run, so how about adding some SQL that contains a guessed name for a table
  - If guess is wrong, get 500 server error
  - If guess is right, well get some kind of response
- Input like  
`x' AND 1=(SELECT COUNT(*) FROM tablename); --`
  - If "tablename" is a table that exists, will get one of those pages saying login details mailed to x@example.com

## Guessing table names ...

- They had to guess several times before they scored – "`tablename=members`" was a winner
- But that merely says "members" is a table that can be read by the server application, it isn't necessarily the table that is being used by the "login" application.
- So, a little more probing
  - They already know the names of some of fields in the table used by login program
  - Try table-name/field-name combinations e.g.  
`x' AND members.email IS NULL; --`

## The users ...

- The members table is "members" with fields like email, full\_name etc;
  - You can probably find some email addresses (or full\_names) given on other pages (e.g. "About us", "Contact us")
  - If you've got some idea of name, but not certain, use the "like" comparison operator in SQL
- Try queries with inputs like `x' OR full_name LIKE '%Bob%'`
- *"The idea is to submit a query that uses the LIKE clause, allowing us to do partial matches of names or email addresses in the database, each time triggering the "We sent your password" message and email.*  
**Warning:** *though this reveals an email address each time we run it, it also actually sends that email, which may raise suspicions. This suggests that we take it easy."*

## Guessing passwords

- Now have valid email identifier, *could go back to login page and try entering email along with guessed passwords*
  - Even dumb sites and dumb systems administrators are likely to spot you
    - Probably show in the log file (sometimes sys admins do look in log files)
    - Code that does password check might be a little smarter
      - Records last login time in database, and last attempted login time along with count of failures
      - If too many failures, may disable account and notify sys admin

This is why sys admins make you pick long contorted passwords like J1Lr8tch07

## Guessing quietly

- Do actual password testing in the injected SQL by including the email name and password directly; e.g. try user **bob@example.com** with multiple passwords.

```
SELECT email, passwd, login_id, full_name FROM members WHERE email = 'bob@example.com' AND passwd = 'hello123';
```
- This is clearly well-formed SQL, so don't get see any server errors
- If do actually guess bob's password, will receive the "your password has been mailed to you" message.
  - Our mark has now been tipped off, but we do have his password.
- This procedure can be automated with scripting in perl

## Modify the database!

- If server-side script can read the data table it can probably write to it!
  - Actually, some databases will allow more subtle permissions settings
    - Could make the members table read-only to user WWW while allowing user WWW to update another table like "orders"
  - If your database makes it easy to be more discriminating in the access that it allows, and if it fits with the logic of your application, then put in as many restrictions as you can!

## Add a new user!

- They tried. Composed an input that would result in server-side script running a query like:

```
SELECT email, passwd, login_id, full_name FROM members WHERE email = 'x';
INSERT INTO members ('email','passwd','login_id','full_name') VALUES ('steve@unixwiz.net','hello','steve','Steve Friedl');--;
```
- It failed

## Reasons for failure

- Who knows!
- Likely reasons:
  1. The web application user might not have **INSERT** permission on the **members** table.
  2. There are undoubtedly other fields in the **members** table, and some may *require* initial values, causing the **INSERT** to fail.
  3. Even if we manage to insert a new record, the application itself might not behave well due to the auto-inserted NULL fields that we didn't provide values for.
  4. A valid "member" might require not only a record in the **members** table, but associated information in other tables (say, "accessrights"), so adding to one table alone might not be sufficient.
- But this approach is obviously worth trying!

## Modify the email field

- **bob@example.com** had an account on the system
- So, use SQL injection to update his database record with our email address:

```
SELECT email, passwd, login_id, full_name FROM members WHERE email = 'x';
UPDATE members SET email = 'steve@unixwiz.net' WHERE email = 'bob@example.com';
```
- After running this, we of course received the "we didn't know your email address", but this was expected due to the dummy email address provided. The **UPDATE** wouldn't have registered with the application, so it executed quietly.

## Use the "I forgot my password" form

- As "bob", ask for password to be mailed – to the newly set address

```
From: system@example.com
To: steve@unixwiz.net
Subject: Intranet login
This email is in response to your request for your
Intranet log in information.
Your User ID is: bob Your password is: hello
```

- Now go back and again edit the table so that your address doesn't stay there; (of course, you did either use an address at an anonymous forwarding service, or a newly created and never again used hotmail account)

## Can now login as a user

- So now can access web system as a user
  - Still may not be able to do much, but you never know your luck
    - If it is a quickly hacked up Microsoft site, then there is a good chance that the web passwords are the same as Windows login passwords.
      - If you are really lucky, you may find you can do a remote login to the Windows system as user bob – and that may allow you to run programs there; FUN!
  - Could possibly get to "interesting" company resources that shouldn't be available to an outsider

## Avoiding such problems

- Use bound parameters in your SQL statements
  - All the carefully crafted attack SQL becomes just part of a name string – a name string that doesn't match anything in the data table
- Limit database permissions on tables
- Use stored procedures
- Review your access logs – watch out for strange patterns of attempted access
- Isolate the web-server
- Insist on hard to guess passwords

## It is not just SQL injection

- Session identifiers
  - Those cookies that you generate as session keys – they aren't just counters are they?
    - If hacker can guess mechanism for generating session keys, he can try to take over someone else's session.
      - Or may try to re-open and continue a session that should have ended

## Threat Modelling

## How are you going to be attacked?

- Primary (attack on your infrastructure)
  - Attacker manages to run a command on your system that shouldn't be run.
    - Common approach is to mix command and data streams in such a way that system processing the data stream will be tricked into running extra commands; e.g. the SQL injection attack
      - Data were email and password
      - Command – extra SQL – was mixed into this data
  - Attacker manages to feed data to valid processing functions on your system that will cause them to execute invalid actions.
- Secondary (attack via network access)
  - Denial of service and similar – attacker disrupts/delays work of legitimate users
  - Attacker can listen to unencrypted data traffic

## What might attacker do?

- "STRIDE" – categories for malicious acts by attacker:
  - Spoofing
    - Attackers effectively usurp identity of valid user
  - Tampering
    - Attackers modify data to their advantage
  - Repudiation
    - Attackers run commands/modify data to their advantage in such a way that these changes can be sourced back to them
  - Information disclosure
    - Attackers read private data
  - Denial of Service
    - Attackers can slow/prevent usage by legitimate users
  - Elevation of privilege
    - Attacker (in principle a legitimate user – e.g. "guest") can run commands/access data that should not be available at his/her trust level
- Most attacks involve actions in more than one of these categories

## Assessing your risk

- "DREAD" – come up with some score that represents the seriousness of an attack; components in score include
  - Damage potential
    - How much will a successful attack cost you?
  - Reproducibility
    - How easy is it to make such an attack successfully?
  - Exploitability
    - How much effort must attacker expend?
  - Affected users
    - What proportion of deployed systems are vulnerable to such an attack?
  - Discoverability
    - How easy would it be for an attacker to discover the vulnerabilities and develop an attack?

## What to do?

- Follow some reasonably systematic way to teasing out problems with your application
  - Identify kind of attack (STRIDE), resources attacked, and attack path
- Evaluate consequences of a successful attack
  - Use that DREAD score – is problem sufficiently serious that you want to devote resources to fixing it
- Using earlier analysis of the problem, find some point on the attack path where additional constraints can be imposed to prevent the attack from succeeding
- Add those constraints



(Not that great a book, but covers some useful concepts in a rather tedious way.)

## Securing your system

- Threat modeling
  - Starts when design for system is pretty much feature complete
    - Identify "entry points", "assets", goals for an attacker
    - Trace data flow through functionality
    - Identify vulnerability
- Code review
  - Starts during implementation phase, review sections of code for the usual errors (buffer overruns, mixing of command and data streams – like SQL injection or data for "system" calls)
  - Peer review of the code (costly)
- Performing penetration tests
  - Starts once an executable is available, testers (who should have access to "threat model" to help them) try to break operations

## Threat modeling

1. Understand adversary's view
  - Entry points
  - Assets
  - "Trust levels"
2. Characterize security of system
  - Use scenarios
  - Assumptions and dependencies
  - Model the system
3. Determine threats
  - Identify threats
  - Determine vulnerabilities

## Understand adversary's view

## Entry points

- Any location where data or commands enter system
  - Web, sockets, rpc/rmi, file-system, ...
    - File-system?
      - E.g. configuration files, LD\_LIBRARY\_PATH, ...
        - It doesn't matter that apparently would need very high privileges to change these (e.g. attacker could feed data to a database stored procedure that can run shell commands – such things exist! – and so change your file system from outside)

## Assets

- Any resources that an adversary might try to modify, corrupt, or just read
  - Attacker must have some reason to attack – even if it is simply to put their "tag" on your web-page.
  - Assets are primarily data (but can be OS data such as entries in process table, sockets, ...)
  - Assets have replacement cost
    - What does a tag on your web page cost? Its cost is in terms of things like customer confidence in the quality of your business and its processes.

## Trust levels

- (Misleading name)
- Legitimate users of a system (external entities) have different usage rights
  - Users provide identifying credentials
  - System determines access rights
    - Access to commands
    - Access to data
- "Trust levels" are a characterization of these external entities and their privileges

## Example

- "Humongous Insurance Price Quote"
    - Typical web application with form data entry and access to database
- Web
- Anonymous user
    - Can apply for account providing requested name/password
  - Account holder
    - Can request insurance quotes – supplying lots of (confidential) data characterising applicant; requested stored in database for processing by an insurance agent
  - Insurance agent
    - Logs in
    - Reads stored request data, makes quote (to be emailed back to applicant)
  - Administrators (web and database)
    - Configures web server and database; creates privileged accounts for the insurance agents

## Trust levels

1. Remote anonymous user
2. Remote user with login credentials (checked)
3. Insurance agent
4. Website administrator
5. Database administrator
6. Web server process identity
7. Database service process identity
8. HTTP user
9. HTTPS user

(HTTP used only to access Humongous's Welcome page; all other web access via HTTPS)

## Entry points

- Rather a lot of these, characterized by name, description, and the (minimum) "Trust level" that permits legitimate use
- Examples:

"Name"	Description	Trust level
Secure web server port HTTPS	Port used by this web-app	Remote anonymous user; remote logged in user; insurance agent; website admin; HTTPS user
Login page	Select new user to create new login account, or login with pre-existing credentials	Remote anonymous user; remote logged in user; insurance agent; HTTPS user

## Entry points - continued

"Name"	Description	Trust level
Create login function	Creates new remote user login	Remote anonymous user;
Login	Checks user supplied credentials and, if ok, begins new session	Remote anonymous user; remote logged in user; insurance agent;
Data entry page	Form supplying details characterizing user	remote logged in user; HTTPS user
Retrieve data function	Form to allow user to review previously entered data (and possibly insurance quote data)	remote logged in user;
Submit data	Forms to submit a request for insurance quote	remote logged in user;

## Entry points - continued

"Name"	Description	Trust level
Insurance agent quote review page	Retrieve a request for quote, submit quote details	Insurance agent; HTTPS user
...	...	...
Database listening port	Enables database use	Database server administrator; web server process identity; database server process identity;
Database stored procedures	Store and retrieve insurance quotes	remote logged in user; insurance agent; database server administrator; web server process identity;
...	...	...

## Entry points - continued

"Name"	Description	Trust level
Web pages on disk	Web pages on disk all entry points	Website administrator; web server process id;
...	...	...
Connection to smtp	Emails sent to user when quotes created by agent	web server process identity;
...	...	...
...	...	...

## Assets

- Data and functionality that must be protected

"Name"	Description	Trust level
User's login data	User-name, password	Remote user with login credentials; database server administrator
Insurance agent's login data	User-name, password	Insurance agent; database server administrator
User's personal data	User data; contact details; data relating to insurance need	Remote user with login credentials; insurance agent;
System	"Assets that relate to underlying system"	
Availability of site	If website down, no work done, no money earned	Website administrator; database administrator
...	...	...

## Assets - continued

"Name"	Description	Trust level
Login session	Web session for user	Remote user with login credentials; insurance agent
...	...	...
Email notification of ready quote	Email sent to user telling them to login and get quote	Remote user with login credentials; insurance agent;
...	...	...
Audit data	Adversaries might try to attack system without being logged	Website administrator; database administrator; Web and database process identifiers
...	...	...

## Characterize security of the system

## Define use scenarios

- Define valid scenarios for use
  - Give some thought to plausible but unsupported use scenarios
- Example – Humungous Price Quote web site
  1. Runs on a web server secured to current industry standards
  2. Uses a database server secured to current industry standards
  3. Web server protected from internet by firewall allowing only HTTP and HTTPS traffic
  4. Communication between web server and database over a private network
  5. Only Welcome page available via HTTP, rest of web access uses HTTPS

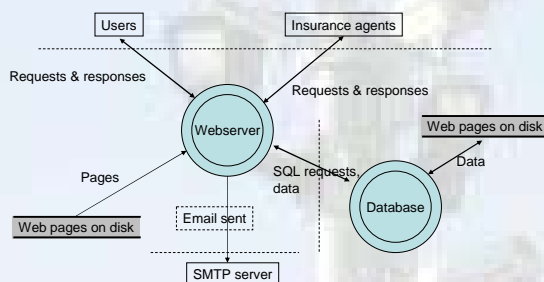
## Identify assumptions and dependencies

- External dependencies
  - E.g. "credit card check service", mail-server, ...
- Example – Humungous web site
  1. Depends on security of its web server
  2. Depends on security of database server
  3. Depends on private network – if this could be monitored data would leak, if accessed then attacks on database possible
  4. Depends on session management – if insecure could get user sessions usurped
  5. Depends on mail server

## Model the system

- Use Data Flow Diagrams (DFDs) to understand the actions that a system performs at a given entry point.
- DFDs can help answer questions such as
  - How are protected resources affected?
  - Where could an external entity manipulate an asset?
  - What processing occurs beyond an entry point?
  - What action is taken on behalf of an external entity, what transformations are applied to the data?

## DFD for Humungous



## Crossing boundaries ...

- Places where cross privilege boundaries (dotted lines on DFD) are often "dangerous"
  - Check carefully for differing assumptions regarding trust levels etc

## Determine threats

## Start with system assets

- How could attacker use asset to
  - Modify control flow
  - Retrieve restricted information
  - Manipulate information
  - Cause system to fail
  - Gain additional rights
- Could attacker access asset
  - Without being audited
  - Skipping access control checks
  - By appearing as another user

## Create and document "threats"

- Threat
  - Entry point(s) used
  - Assets that the threat targets
  - STRIDE classification (effects of threat, usually more than one STRIDE category – spoof, tamper, ...)
  - Identification data for subsequent reference
  - Mitigation
    - Once threat identified, work out how one might reduce severity or prevent

## Example threats to Humungous system: 1

- Malicious SQL data in input
  - Attacker using SQL injection
  - STRIDE: Tampering, elevation of privilege
  - Mitigation
    - ? Not yet anyway
  - Entry points
    - Login page, data entry page, insurance agent review page
  - Asset
    - Database tables
  - Notes
    - They found that their code did build a SQL request from unchecked user input (DOH!)

## Example threats to Humungous system: 2

- Disclosure of login information
  - Attacker gets username and password of other user
  - STRIDE: Information disclosure, elevation of privilege
  - Mitigation
    - Maybe not
    - Approaches to mitigation:
      - Firewall to protect database from remote access (see scenarios)
      - Enforce requirement for strong passwords so brute force guessing impractical
  - Entry points
    - Login page, database access (to tables and/or stored procedures)
  - Asset
    - User / insurance-agent login details

## Example threats to Humungous system: 3

- Session ID theft
  - Attacker (maybe using network traffic sniffer or just by guessing) gets session id of another user
  - STRIDE: Elevation of privilege
  - Mitigation
    - Yes
      - Session ids generated by secure web-server session management code (so shouldn't be guessable)
        - But that is an "external dependency" because if the session management algorithm isn't good, then ids would be guessable
      - HTTPS used for all traffic so shouldn't be able to snoop
        - See the scenarios
  - Entry points
    - Web server port
  - Asset
    - Login session

### Example threats to Humungous system: 3

- Quote tampering
  - Attacker modifies the quote that an insurance agent created and left on database for viewing by user
  - STRIDE: Tampering
  - Mitigation
    - Yes
      - (later example)
  - Entry points
    - Insurance agent quote review page; database stored procedure
  - Asset
    - Accuracy of price quote

### Example threats

- Something like 14 threats identified for this web site
- Many will be "standard" for any web-based application
  - So shouldn't cost too much to identify and review and there should be "standard" ways of mitigating them.

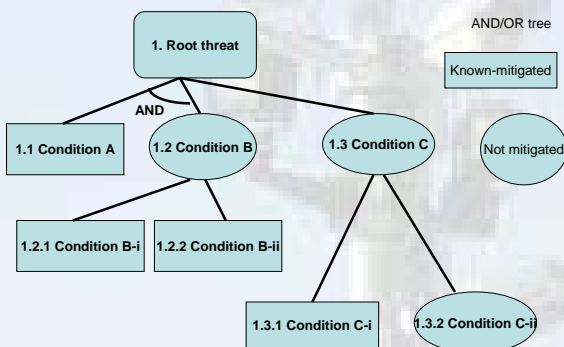
### Why document threats that are mitigated?

- The example – "Session id threat" : "Mitigated"
  - (It's solved by standard web application security approaches)
- So why bother considering it and documenting it?
  1. Can get a better idea of how "secure" an application is if do consider all threats and then identify ratio of mitigated to unmitigated
  2. It may not stay mitigated – mitigation assumed use of HTTPS and a good session key generator algorithm, someone might think to downgrade things in future

### Next step: "Threat Trees"

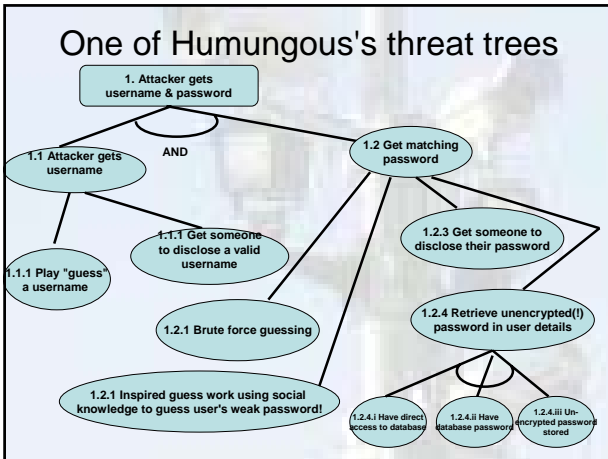
- If cannot readily establish that threat is mitigated, need to explore issue further.
  - Whether it is real or is actually mitigated has to be determined by working out conditions that must apply for attacker to succeed
  - Exploration represented graphically via "threat tree"
- Threat tree
  - Root node represents threat
  - Subordinate nodes (in and/or tree structure) represent conditions that would have to apply for attacker to achieve threat

### Generic threat tree



### Generic threat tree

- Root threat
  - Attacker must
    - Achieve condition 1.1 and 1.2; or achieve condition 1.3
- Attack via 1.1 and 1.2
  - To achieve condition 1.2, attacker must achieve 1.2.1 or 1.2.2
    - But know mitigation approaches prevent either of these being attained
    - So while condition 1.2 itself has no constraints on it, we know attacker cannot achieve this condition
  - In any case, we know methods of preventing condition 1.1 from being achieved
  - Therefore this attack path is blocked
- Attack via 1.3
  - This is open,
  - While we can block the attacker from route via 1.3-i, there seems to be an attack 1.3-ii, 1.3, 1
- **Vulnerability exists!**



- ### Determine vulnerabilities
- Check those trees
    - Is there a complete path of from "node" conditions to "root threat" that is unmitigated?
    - If yes, you have a vulnerability.
  - Compute a DREAD score for the vulnerability
    - Try to quantify (quite crudely) damage potential, exploitability, etc
  - Based on DREAD score and estimated cost of finding mitigation approach
    - Live with vulnerability
    - Fix it

- ### Humungous : discovered vulnerabilities : 1
- RetrieveCredentials SQL injection
    - STRIDE: Tampering, Elevation of privilege
    - DREAD: "10" (i.e. do something about it)
    - Corresponding threat: threat 1, SQL injection into username and bad SQL statement preparation
  - Fixing it:
    - "Sanitize" inputs, use prepared statements

- ### Humungous : discovered vulnerabilities : 2
- Username discoverability
    - STRIDE: Information disclosure, elevation of privilege
    - DREAD: "6.6" (i.e. probably should do something about it)
    - Corresponding threat: threat 2, discovery of username and password
      - Particular issue was error responses distinguished "Invalid username" and "Incorrect password" rather than just "Invalid credentials" – making it relatively easy to guess at least the username
  - Fixing it:
    - See CWE209!

- ### Humungous : discovered vulnerabilities : ... 5
- Site has no logging or auditing
    - STRIDE: Repudiation
    - DREAD: "5.2" (i.e. probably should do something about it)
    - Corresponding threat: threat 15, adversary tried to access other user's data without being logged
  - Fixing it:
    - Determine what functionality should be logged and how logs should be audited

### Threat modeling

## Just another helper script

- Much like RUP which provided "scripts" of activities you should engage in when filling specific role at some phase in software development process
- Gives some ideas as to how to check through an application
  - Does apply to things other than web sites as well
- Bit tedious
  - As always, a lot of this stuff becomes easier with experience "This website is likely to have similar problems to last one, so ..."