1. Welcome Back!
2. Dominic – Please take minutes?
3. Team Website – John Lupton
   a. Suggested content areas:
      i. Meeting Agendas & Meeting Minutes
      ii. Tasks - Who is working on what
      iii. Work In Progress - documents in draft
      iv. Web App Security Team Deliverables - these would be pdfs
      v. Blog
      vi. Miscellaneous
   b. Other areas?
   c. Process for updating documents; PennKey authentication; other?

4. Discuss the tasks/updates team members created for A1 Unvalidated Input (See 12/1/05 meeting minutes for details.)
   a. Section A1.1 Description - Susan Kennedy to write up additional risks associated with unvalidated input.
   b. Section A1.2 Environments Affected - Steve Kratowicz to make a grid of available tools for protecting against malicious input by development platform.
   c. Section A1.4 - How to Determine If You Are Vulnerable - Terry Ryan to write up an Cold Fusion example and Ray Bokenkamp will write up a Php example.
   d. Section A1.5 How To Protect Yourself – Everyone to develop 1-2 examples.

5. OWASP #2 – Broken Access Control (25 minutes)
6. OWASP #3 – Broken Authentication and Session Management (If time permits)

Current OWASP Wording:
A1 Unvalidated Input

A1.1 Description

Web applications use input from HTTP requests (and occasionally files) to determine how to respond. Attackers can tamper with any part of an HTTP request, including the url, querystring, headers, cookies, form fields, and hidden fields, to try to bypass the site’s security mechanisms. Common names for common input tampering attacks include: forced browsing, command insertion, cross site scripting, buffer overflows, format string attacks, SQL injection, cookie poisoning, and hidden field manipulation. Each of these attack types is described in more detail later in this paper.

- A4 – Cross Site Scripting Flaws discusses input that contains scripts to be executed on other user’s browsers
- A5 – Buffer Overflows discusses input that has been designed to overwrite program execution space
- A6 – Injection Flaws discusses input that is modified to contain executable commands

Some sites attempt to protect themselves by filtering out malicious input. The problem is that there are so many different ways of encoding information. These encoding formats are not like encryption, since they are trivial to decode. Still, developers often forget to decode all parameters to their simplest form before using them. Parameters must be converted to the simplest form before they are validated, otherwise, malicious input can be masked and it can slip past filters. The process of simplifying these encodings is called “canonicalization.” Since almost all HTTP input can be represented in multiple formats, this technique can be used to obfuscate any attack targeting the vulnerabilities described in this document. This makes filtering very difficult.

A surprising number of web applications use only client-side mechanisms to validate input. Client side validation mechanisms are easily bypassed, leaving the web application without any protection against malicious parameters. Attackers can generate their own HTTP requests using tools as simple as telnet. They do not have to pay attention to anything that the developer intended to happen on the client side. Note that client side validation is a fine idea for performance and usability, but it has no security benefit whatsoever. Server side checks are required to defend against parameter manipulation attacks. Once these are in place, client side checking can also be included to enhance the user experience for legitimate users and/or reduce the amount of invalid traffic to the server.

A1.2 Environments Affected

All web servers, application servers, and web application environments are susceptible to parameter tampering.

Susan to draft wording on risks here

A1.3 Examples and References

- modsecurity project (Apache module for HTTP validation) http://www.modsecurity.org
- How to Build an HTTP Request Validation Engine (J2EE validation with Stinger) http://www.owasp.org/columns/jeffwilliams/jeffwilliams2
- Have Your Cake and Eat it Too (.NET validation) http://www.owasp.org/columns/jpoteet/jpoteet2
A1.4 How to Determine If You Are Vulnerable

Any part of an HTTP request that is used by a web application without being carefully validated is known as a “tainted” parameter. The simplest way to find tainted parameter use is to have a detailed code review, searching for all the calls where information is extracted from an HTTP request. For example, in a J2EE application, these are the methods in the HttpServletRequest class. Then you can follow the code to see where that variable gets used. If the variable is not checked before it is used, there is very likely a problem. In Perl, you should consider using the “taint” (-T) option.

It is also possible to find tainted parameter use by using tools like OWASP’s WebScarab. By submitting unexpected values in HTTP requests and viewing the web application’s responses, you can identify places where tainted parameters are used.

Ray/Terry to draft examples here

A1.5 How to Protect Yourself

The best way to prevent parameter tampering is to ensure that all parameters are validated before they are used. A centralized component or library is likely to be the most effective, as the code performing the checking should all be in one place. Each parameter should be checked against a strict format that specifies exactly what input will be allowed. “Negative” approaches that involve filtering out certain bad input or approaches that rely on signatures are not likely to be effective and may be difficult to maintain.

Parameters should be validated against a “positive” specification that defines:

- Data type (string, integer, real, etc…)
- Allowed character set
- Minimum and maximum length
- Whether null is allowed
- Whether the parameter is required or not
- Whether duplicates are allowed
- Numeric range
- Specific legal values (enumeration)
- Specific patterns (regular expressions)

A new class of security devices known as web application firewalls can provide some parameter validation services. However, in order for them to be effective, the device must be configured with a strict definition of what is valid for each parameter for your site. This includes properly protecting all types of input from the HTTP request, including URLs, forms, cookies, querystrings, hidden fields, and parameters.

The OWASP Filters project is producing reusable components in several languages to help prevent many forms of parameter tampering. The Stinger HTTP request validation engine (stinger.sourceforge.net) was also developed by OWASP for J2EE environments.

Everyone to draft examples here

OWASP #2 -
A2 Broken Access Control

A2.1 Description

Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others. These checks are performed after authentication, and govern what ‘authorized’ users are allowed to do. Access control sounds like a simple problem but is insidiously difficult to implement correctly. A web application’s access control model is closely tied to the content and functions that the site provides. In addition, the users may fall into a number of groups or roles with different abilities or privileges.

Developers frequently underestimate the difficulty of implementing a reliable access control mechanism. Many of these schemes were not deliberately designed, but have simply evolved along with the web site. In these cases, access control rules are inserted in various locations all over the code. As the site nears deployment, the ad hoc collection of rules becomes so unwieldy that it is almost impossible to understand.

Many of these flawed access control schemes are not difficult to discover and exploit. Frequently, all that is required is to craft a request for functions or content that should not be granted. Once a flaw is discovered, the consequences of a flawed access control scheme can be devastating. In addition to viewing unauthorized content, an attacker might be able to change or delete content, perform unauthorized functions, or even take over site administration.

One specific type of access control problem is administrative interfaces that allow site administrators to manage a site over the Internet. Such features are frequently used to allow site administrators to efficiently manage users, data, and content on their site. In many instances, sites support a variety of administrative roles to allow finer granularity of site administration. Due to their power, these interfaces are frequently prime targets for attack by both outsiders and insiders.

A2.2 Environments Affected

All known web servers, application servers, and web application environments are susceptible to at least some of these issues. Even if a site is completely static, if it is not configured properly, hackers could gain access to sensitive files and deface the site, or perform other mischief.

A2.3 Examples and References


A2.4 How to Determine If You Are Vulnerable

Virtually all sites have some access control requirements. Therefore, an access control policy should be clearly documented. Also, the design documentation should capture an approach for enforcing this policy. If this documentation does not exist, then a site is likely to be vulnerable.

The code that implements the access control policy should be checked. Such code should be well structured, modular, and most likely centralized. A detailed code review should be performed to validate the correctness of the access control implementation. In addition, penetration testing can be quite useful in determining if there are problems in the access control scheme.

Find out how your website is administrated. You want to discover how changes are made to webpages, where they are tested, and how they are transported to the production server. If administrators can make changes remotely, you want to know how those communications channels are protected. Carefully review each interface to make sure that only authorized administrators are allowed access. Also, if there are different types or groupings of data that
can be accessed through the interface, make sure that only authorized data can be accessed as well. If such interfaces employ external commands, review the use of such commands to make sure they are not subject to any of the command injection flaws described in this paper.

A2.5 How to Protect Yourself

The most important step is to think through an application’s access control requirements and capture it in a web application security policy. We strongly recommend the use of an access control matrix to define the access control rules. Without documenting the security policy, there is no definition of what it means to be secure for that site. The policy should document what types of users can access the system, and what functions and content each of these types of users should be allowed to access. The access control mechanism should be extensively tested to be sure that there is no way to bypass it. This testing requires a variety of accounts and extensive attempts to access unauthorized content or functions.

Some specific access control issues include:

- Insecure Id’s – Most web sites use some form of id, key, or index as a way to reference users, roles, content, objects, or functions. If an attacker can guess these id’s, and the supplied values are not validated to ensure the are authorized for the current user, the attacker can exercise the access control scheme freely to see what they can access. Web applications should not rely on the secrecy of any id’s for protection.

- Forced Browsing Past Access Control Checks – many sites require users to pass certain checks before being granted access to certain URLs that are typically ‘deeper’ down in the site. These checks must not be bypassable by a user that simply skips over the page with the security check.

- Path Traversal – This attack involves providing relative path information (e.g., “../../../target_dir/target_file”) as part of a request for information. Such attacks try to access files that are normally not directly accessible by anyone, or would otherwise be denied if requested directly. Such attacks can be submitted in URLs as well as any other input that ultimately accesses a file (i.e., system calls and shell commands).

- File Permissions – Many web and application servers rely on access control lists provided by the file system of the underlying platform. Even if almost all data is stored on backend servers, there are always files stored locally on the web and application server that should not be publicly accessible, particularly configuration files, default files, and scripts that are installed on most web and application servers. Only files that are specifically intended to be presented to web users should be marked as readable using the OS’s permissions mechanism, most directories should not be readable, and very few files, if any, should be marked executable.

- Client Side Caching – Many users access web applications from shared computers located in libraries, schools, airports, and other public access points. Browsers frequently cache web pages that can be accessed by attackers to gain access to otherwise inaccessible parts of sites. Developers should use multiple mechanisms, including HTTP headers and meta tags, to be sure that pages containing sensitive information are not cached by user’s browsers.

There are some application layer security components that can assist in the proper enforcement of some aspects of your access control scheme. Again, as for parameter validation, to be effective, the component must be configured with a strict definition of what access requests are valid for your site. When using such a component, you must be careful to understand exactly what access control assistance the component can provide for you given your site’s security policy, and what part of your access control policy that the component cannot deal with, and therefore must be properly dealt with in your own custom code.

For administrative functions, the primary recommendation is to never allow administrator access through the front door of your site if at all possible. Given the power of these interfaces, most organizations should not accept the risk of making these interfaces available to outside attack. If remote administrator access is absolutely required, this can be accomplished without opening the front door of the site. The use of VPN technology could be used to provide an outside administrator access to the internal company (or site) network from which an administrator can then access the site through a protected backend connection.
A3 Broken Authentication and Session Management

A3.1 Description
Authentication and session management includes all aspects of handling user authentication and managing active sessions. Authentication is a critical aspect of this process, but even solid authentication mechanisms can be undermined by flawed credential management functions, including password change, forgot my password, remember my password, account update, and other related functions. Because "walk by" attacks are likely for many web applications, all account management functions should require reauthentication even if the user has a valid session id.

User authentication on the web typically involves the use of a userid and password. Stronger methods of authentication are commercially available such as software and hardware based cryptographic tokens or biometrics, but such mechanisms are cost prohibitive for most web applications. A wide array of account and session management flaws can result in the compromise of user or system administration accounts. Development teams frequently underestimate the complexity of designing an authentication and session management scheme that adequately protects credentials in all aspects of the site.

Web applications must establish sessions to keep track of the stream of requests from each user. HTTP does not provide this capability, so web applications must create it themselves. Frequently, the web application environment provides a session capability, but many developers prefer to create their own session tokens. In either case, if the session tokens are not properly protected, an attacker can hijack an active session and assume the identity of a user. Creating a scheme to create strong session tokens and protect them throughout their lifecycle has proven elusive for many developers.

Unless all authentication credentials and session identifiers are protected with SSL at all times and protected against disclosure from other flaws, such as cross site scripting, an attacker can hijack a user's session and assume their identity.

A3.2 Environments Affected
All known web servers, application servers, and web application environments are susceptible to broken authentication and session management issues.

A3.3 Examples and References

A3.4 How to Determine If You Are Vulnerable
Both code review and penetration testing can be used to diagnose authentication and session management problems. Carefully review each aspect of your authentication mechanisms to ensure that user's credentials are protected at all times, while they are at rest (e.g., on disk), and while they are in transit (e.g., during login). Review every available mechanism for changing a user's credentials to ensure that only an authorized user can change them. Review your session management mechanism to ensure that session identifiers are always protected and are used in such a way as to minimize the likelihood of accidental or hostile exposure.

A3.5 How to Protect Yourself
Careful and proper use of custom or off the shelf authentication and session management mechanisms should significantly reduce the likelihood of a problem in this area. Defining and documenting your site's policy with respect to securely managing users credentials is a good first step. Ensuring that your implementation consistently enforces
this policy is key to having a secure and robust authentication and session management mechanism. Some critical areas include:

- **Password Strength** - passwords should have restrictions that require a minimum size and complexity for the password. Complexity typically requires the use of minimum combinations of alphabetic, numeric, and/or non-alphanumeric characters in a user’s password (e.g., at least one of each). Users should be required to change their password periodically. Users should be prevented from reusing previous passwords.

- **Password Use** - Users should be restricted to a defined number of login attempts per unit of time and repeated failed login attempts should be logged. Passwords provided during failed login attempts should not be recorded, as this may expose a user’s password to whoever can gain access to this log. The system should not indicate whether it was the username or password that was wrong if a login attempt fails. Users should be informed of the date/time of their last successful login and the number of failed access attempts to their account since that time.

- **Password Change Controls**: A single password change mechanism should be used wherever users are allowed to change a password, regardless of the situation. Users should always be required to provide both their old and new password when changing their password (like all account information). If forgotten passwords are emailed to users, the system should require the user to reauthenticate whenever the user is changing their e-mail address, otherwise an attacker who temporarily has access to their session (e.g., by walking up to their computer while they are logged in) can simply change their e-mail address and request a ‘forgotten’ password be mailed to them.

- **Password Storage** - All passwords must be stored in either hashed or encrypted form to protect them from exposure, regardless of where they are stored. Hashed form is preferred since it is not reversible. Encryption should be used when the plaintext password is needed, such as when using the password to login to another system. Passwords should never be hardcoded in any source code. Decryption keys must be strongly protected to ensure that they cannot be grabbed and used to decrypt the password file.

- **Protecting Credentials in Transit** - The only effective technique is to encrypt the entire login transaction using something like SSL. Simple transformations of the password such as hashing it on the client prior to transmission provide little protection as the hashed version can simply be intercepted and retransmitted even though the actual plaintext password might not be known.

- **Session ID Protection** – Ideally, a user’s entire session should be protected via SSL. If this is done, then the session ID (e.g., session cookie) cannot be grabbed off the network, which is the biggest risk of exposure for a session ID. If SSL is not viable for performance or other reasons then session IDs themselves must be protected in other ways. First, they should never be included in the URL as they can be cached by the browser, sent in the referer header, or accidentally forwarded to a ‘friend’. Session IDs should be long, complicated, random numbers that cannot be easily guessed. Session IDs can also be changed frequently during a session to reduce how long a session ID is valid. Session IDs must be changed when switching to SSL, authenticating, or other major transitions. Session IDs chosen by a user should never be accepted.

- **Account Lists** - Systems should be designed to avoid allowing users to gain access to a list of the account names on the site. If lists of users must be presented, it is recommended that some form of pseudonym (screen name) that maps to the actual account be listed instead. That way, the pseudonym can’t be used during a login attempt or some other hack that goes after a user’s account.

- **Browser Caching** – Authentication and session data should never be submitted as part of a GET, POST should always be used instead. Authentication pages should be marked with all varieties of the no cache tag to prevent someone from using the back button in a user’s browser to backup to the login page and resubmit the previously typed in credentials. Many browsers now support the autocomplete=false flag to prevent storing of credentials in autocomplete caches.

- **Trust Relationships** – Your site architecture should avoid implicit trust between components whenever possible. Each component should authenticate itself to any other component it is interacting with unless there is a strong reason not to (such as performance or lack of a usable mechanism). If trust relationships are required, strong procedural and architecture mechanisms should be in place to ensure that such trust cannot be abused as the site architecture evolves over time.