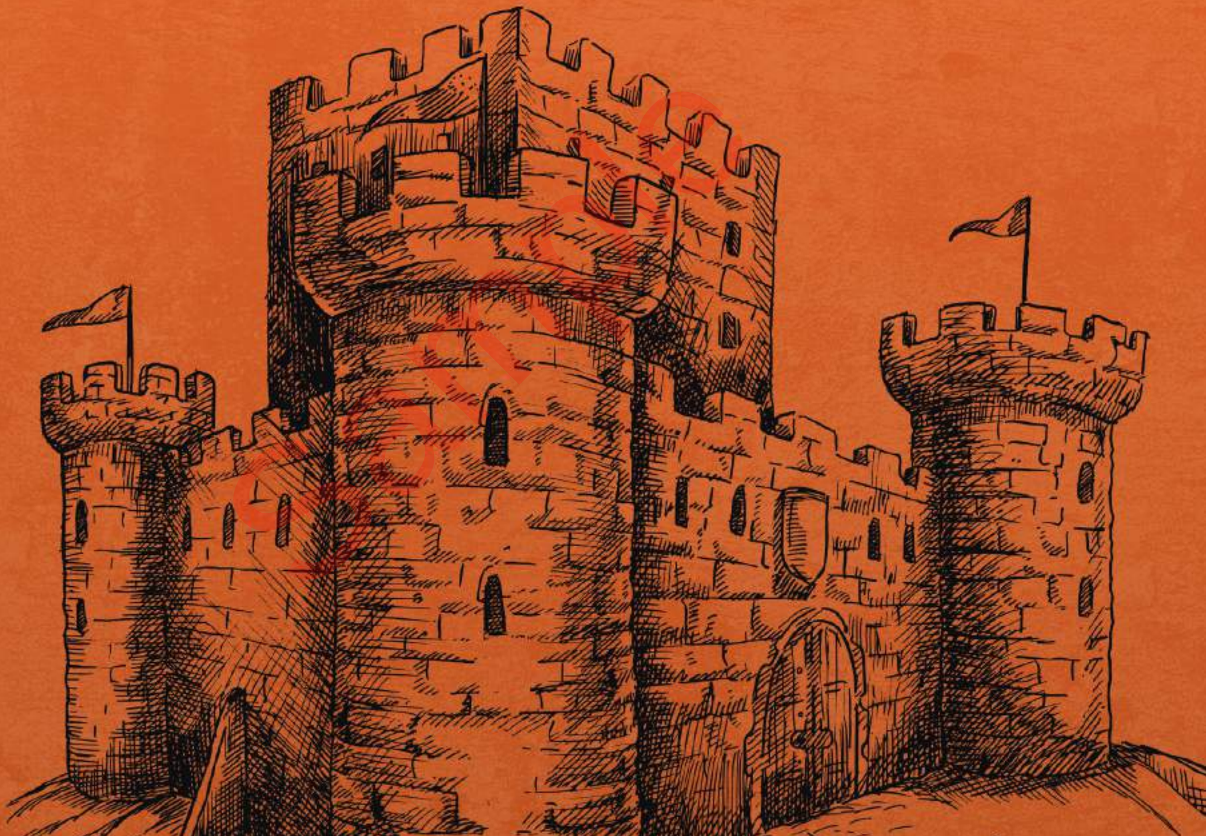
Security Principles for PHP Applications

by Eric Mann



a php[architect] guide

Table of Contents

1	**Application Security From First Principles**	**1**
	The Common Mistake	2
	An Example of Broken Standards Implementation—JOSE	3
	A Security-First Mindset	4
	Accurate Threat Models	5
	Looking Ahead	6
2	**About This Book**	**7**
	Who This Book Is For	7
	How to Use This Book	8
	Code Examples	8
3	**OWASP**	**11**
	The OWASP Top Ten	11
	The Risk of Lists	14

4 ASR1: Injection 15
How Big of a Deal Is This? 16
How Would This Look in Production? 17
How to Prevent These Vulnerabilities 23
Conclusion 29

5 ASR2: Broken Authentication and Session Management 31
Issues Facing Authentication 32
How Could Each of These Be Fixed? 38
Conclusion 44

6 ASR3: Sensitive Data Exposure 45
What Are Some of the Practical Risks to Sensitive Data? 46
How Can These Risks Be Effectively Mitigated? 51
Conclusion 56

7 ASR4: XML External Entities (XXE) 57
How an Application Can Be Exploited 58
How Do We Prevent Loading External Elements? 59
How Do We Prevent Expanding Elements? 60
Conclusion 60

8 ASR5: Broken Access Control 61
How Would This Look in Production? 62
Has This Ever Happened? 64
How Would This Code Look If Patched? 65
Conclusion 67
What Did United Airlines Do? 67

9 ASR6: Security Misconfiguration 69
How Would This Look in Production? 70
How Would This Code Look If Patched? 74
Conclusion 79

10 ASR7: Cross-Site Scripting (XSS) 81
How Would This Look in Production? 82
How Would This Code Look If Patched? 86
Conclusion 89

11 ASR8: Insecure Deserialization 91
Object Injection Vulnerabilities 92
DoS Vulnerabilities 95
Potential Production-Ready Solutions 96
Conclusion 97

12 ASR9: Using Components With Known Vulnerabilities 99
What Does This Look Like in Code? 100
Are Libraries the Only Risk? 102
PHP as a Root Dependency 104
How Do You Protect Yourself? 104
Auditing the Entire Application Stack 105
Conclusion 106

13 ASR10: Insufficient Logging and Monitoring 107

Why Logging Matters 108

What Events Should We Log? 108

What Data Should We Log? 111

How Should We Log Data? 112

How Much Logging Is Too Much? 112

Conclusion 113

14 Keeping Ahead of the Trends 117

A Living Standard 118

15 Insufficient Attack Prevention 121

How Would This Look in Production? 122

Request Size 123

In the Wild: WordPress XML-RPC Vulnerability 124

What Can I Do About It? 126

Conclusion 131

16 Underprotected APIs 133

What Are Some of the Potential Vulnerabilities? 134

How Can These Be Prevented? 137

17 Cross-Site Request Forgery (CSRF) 147

How Would This Look in Production? 148

How Could This Be Prevented? 151

How Do the Various PHP Frameworks Handle CSRF? 154

Conclusion 159

18 Unvalidated Redirects and Forwards 161
How Would This Look in Production? 162
How Would This Code Look If Patched? 165

19 Peer Code Review 169
Red Teaming 172

20 Further Reading and Resources 175
Static Code Analysis 175
PHP_CodeSniffer 176
Paid Utilities 176
Security Audits 177
The PHP Community 178
Mailing Lists and Feeds 178
Blogs and Resources 179
Conferences and Workshops 179

21 Responsible Disclosure 181
How to Disclose 182
How to Handle Disclosure 183

Index 185

Chapter 4

ASR1: Injection

Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

The risk of injection is one of the most common and well-known vulnerabilities in application development. From a high level, injection attacks happen when an attacker has the ability to control the input of your program. If they can write directly to a database, such that their nefarious data is passed unfiltered, they can inject whatever control systems they wish.

A common injection vulnerability is passing parameters directly from the `$_POST` superglobal, which is user-controlled, into a SQL statement:

VULNERABLE

```
// a common SQL injection vulnerabilty
$name = $_POST['name'];
$sql = "SELECT * FROM users WHERE email='$name'";
$result = $db->query($sql);
```

Assume for a moment this code is meant to look up ticket information for a given user attending a conference. A regular form submission would send the user's email to the server and return a row from the database representing that ticket. Consider instead what would happen if an attacker were to trigger the following cURL request:

```
curl -X POST -d "name=a@b.com' OR 1=1;--" http://yoursite.com
```

The server would accept this value and happily concatenate it into the SQL query, generating the following statement:

```
SELECT * FROM users WHERE email='a@b.com' OR 1=1;--'
```

The `OR` in this query will always evaluate to true, and the `--` at the end forces any content following the broken query to be treated as a comment. Instead of returning a specific user's data, this new query will return the entire users table from the database! The attacker now has all of your attendees' information and can do whatever they wish with it. Further, an attacker could inject any query following this same pattern, potentially injecting, modifying, or even *deleting* data.

In the PHP world, injection like this occurs when developers erroneously trust user input. The vulnerable code above allowed users direct input into SQL queries, making the database do something other than it was intended. Other users can manipulate query variables that are used internally to switch application logic from one, expected flow to another. Still, other users might inject executable PHP code into a header that is extracted and inadvertently executed by the application, giving this user control over the PHP stack itself.

Said another way, injection is when you, the developer, give a user the power to dictate what code is being executed. You're then running their application, and they can do whatever they want. They can dump sensitive data to output. They can write extraneous files to disk. They can insert malicious information into the database. The sky is the limit.

At a minimum, allowing users to download, install, and execute arbitrary scripts could let them fill your server's hard disk with junk. The worst-case scenarios, however, are far more chilling. Among other things, an attacker could:

- Use your server as part of a network to launch a denial-of-service attack against someone else.
- Send spam or phishing emails to third parties.
- Use your server as a proxy or host for other illegal activities.

Abdicating control over the behavior of your application to arbitrary users gives those users a great deal of power; at the end of the day, though, *you* are still ultimately responsible for what your server does.

How Big of a Deal Is This?

It's easy as a developer to discount injection as a serious risk to your application. Often, injection vulnerabilities are reported as the ability for a rogue user to input garbage into an application—the easiest response to such a report is to shrug it off as "garbage in, garbage out."

> **Note:** *It's equally easy to discount injection vulnerabilities applying only to trusted admin or superuser access. Many developers will, mistakenly, assume the only users who ever have access to these accounts are "trusted" in the first place. However, if the application ever exposes a privilege escalation vulnerability, or one of these privileged users is tricked into running a malicious command, the consequences to your application could be huge. The chapters on ASR5: Broken Access Control and Cross-Site Request Forgery have deeper explanations of each issue.*

These reports of garbage being inserted into database fields often come from researchers using tools to "fuzz test"[1] your application. Fuzz testing is the practice of providing broken, unexpected, or purely random input to an application to see what happens. With binary, non-memory-safe applications this is an excellent way to test the handling of invalid input.

Are strings accepted in place of integer inputs? What happens when I pass a control character to a function which otherwise takes benign input? Can I make the application do something unexpected? Can I use this behavior to manipulate the application into doing something other than what was intended?

In some situations, though, the garbage input does make the application behave in ways it's not supposed to. The popular webcomic, xkcd.com, illustrates various security principles on occasion. In this instance, the danger of allowing user input into a SQL statement, see *Exploits of a Mom*[2].

The ability to inject non-alphanumeric characters into a SQL statement makes it trivial to inject your own queries into an otherwise trusted framework. An attacker can `SELECT` data to which they'd otherwise lack access. Another attacker could insert themselves into a list of "administrator" users in the database and take control of the system. Yet another attacker could merely destroy the data upon which your application relies.

Further, not protecting against certain character sets can negatively impact your users down the road. Consider users with names containing apostrophes ("O'Malley" or similar) or non-Latin characters. Any of these could potentially break a SQL statement if not properly escaped.

Injection attacks happen frequently in the wild, most frequently when developers are using unparameterized SQL queries or otherwise passing untrusted user input into executable environments. They give the user (or an attacker) a level of control over the system equivalent to the application itself.

How Would This Look in Production?

Attackers can inject their code into your application in three different ways:

1. They can inject additional queries into a SQL statement.
2. They can render malicious user-submitted input through a form (or query or header) that is then used directly in PHP. This also allows cross-site scripting attacks, covered in detail later

[1] *"fuzz test": http://phpa.me/wikipedia-fuzzing*
[2] *Exploits of a Mom: https://xkcd.com/327/*

in the chapter on *CSRF*.

3. They can upload an executable script which is later invoked through another exposed vulnerability.

The code exposing these vulnerabilities looks slightly different in each case, but all have the same root characteristic: the code trusts user input to fall within certain bounds. It also fails to validate the input or those bounds.

SQL Injection

An older WordPress plugin I built suffered from an injection-related flaw somewhat recently. While I was trying to do my best to protect code from untrusted user input, I mistakenly assumed certain parameters were escaped that, in fact, were not.

The code in question had two fatal flaws. The code was trusting data stored within user-provided cookies; in this case, it trusted it had generated a session ID stored within a cookie itself.

In the application's session controller, the following constructor would grab a predefined cookie and extract various data from it. The code assumes the first part of the cookie is a valid session ID and stores it in the controller for later use.

Listing 4.1 VULNERABLE

```
protected function __construct() {
   if (isset($_COOKIE[WP_SESSION_COOKIE])) {
      $cookie = stripslashes($_COOKIE[WP_SESSION_COOKIE]);
      $cookie_crumbs = explode('||', $cookie);

      $this->session_id = $cookie_crumbs[0];
      $this->expires = $cookie_crumbs[1];
      $this->exp_variant = $cookie_crumbs[2];

      // Update the session expiration if we're past the variant time
      if (time() > $this->exp_variant) {
         $this->set_expiration();
         delete_option("_wp_session_expires_{$this->session_id}");
         add_option("_wp_session_expires_{$this->session_id}",
                    $this->expires, '', 'no');
      }
   } else {
      $this->session_id = WP_Session_Utils::generate_id();
      $this->set_expiration();
   }

   $this->read_data();
   $this->set_cookie();
}
```

Index

A

access control

Access-Control-Allow-Origin, 156

maintaining fine-grained, 69

role-based, 68

violations, 116

addslashes, 21–22

algorithms

asymmetric, 37

decryption, 53

default bcrypt, 44

secure PBKDF, 140

standard AES-256 encryption, 56

Amazon, RDS, 56, 75

Amazon, S3, 79

Amazon. WAF, 129

API, 37, 122–23, 128, 131–33, 135–40, 142, 144–47, 156, 174

authentication, 138

managing keys, 55

REST, 135

application

container, 64–65

errors, 114

logs, 117

mobile, 66

passwords, 144–45

production, 105

risks, 135, 164

security risks (ASRs), 1, 8, 12–14, 31, 47, 122, 130, 139, 183

security team, 184

server, 40, 62, 71, 96, 113, 124

state, 5, 66, 113

ASR1, 17–18, 20, 22, 26, 28, 30, 77, 84, 122, 138

ASR2, 33–34, 36, 38, 40, 42, 44, 46, 58, 122

ASR3, 47–48, 50, 52, 54, 56, 58, 72, 122

ASR4, 59–60, 62, 122

ASR5, 19, 63–64, 66, 68, 122

ASR6, 31, 49, 55, 71–72, 74, 76, 80, 122

ASR7, 83–84, 86, 88, 90, 122, 150

ASR8, 95–96, 98, 100, 122

ASR9, 22, 103–4, 106, 108, 110, 122

ASR10, 111–12, 114, 116, 122

ASRs. See application security risks

attack

cross-domain, 150

phishing, 165

protection, 123

replay, 155

auditing, 109, 111

authentication, 33, 37, 39, 46, 49, 52, 56, 63, 66, 68, 76, 78, 136, 138, 143–45

basic, 5, 138

multi-factor, 138

strong, 151

authentication function, 39

authorization, 46, 63, 68–69, 114, 140, 149, 151

B

bindParam, 151, 155

Bitbucket, 53, 173–74

blacklist, 28, 167

Broken Access Control, 12, 19, 63–64, 66, 68, 122

BruteProtect, 143

Bug bounty programs, 66

C

Cargill, Tom, 1

Cloudflare, 31, 129–30

Codacy, 179

code

  complexity, 179

  dependencies, 109

  deploying, 53

  proof-of-concept, 184

  review, 122, 171, 173–74, 177

  standards, 179

Code Climate, 179

codex.wordpress.org, 68, 157

commands, arbitrary, 28–29, 31

Composer, 3, 108–9, 129, 145

  lockfile, 106

content security policy (CSP), 113

credentials, 2, 48–49, 54–55, 63, 81, 145

  managing, 48

  shared, 49, 54

credit card, 34, 47, 50, 67

Cross-Origin Resource Sharing (CORS), 152

Cross-Site Request Forgery (see CSRF)

Cross-Site Scripting, 12, 83–84, 86, 88, 90, 122, 150, 184

CSRF, 20, 149–50, 152, 154–60

  attacks, 152–53, 156, 160–61

  Guard, 157–58

  invalid token, 155

  PHP Frameworks, 156–57, 159

  protection, 153–54, 156

  tokens, 154, 157–58

curl, 18, 22, 99, 167

D

database

  client applications, 56

  multiple servers, 49

  MySQL, 49, 87

  users, 81

decrypt, 41, 52, 56–58, 140, 144

denial-of-service, 18, 31, 79, 95, 129, 133

deserialization, 100–101

disclosure

  major security, 160

  responsible, 69, 183–85

Drupal, 3, 68–69, 104, 160, 178

  forms API, 157, 160

  hooks, 29

  security team, 181

E

email, 5, 17–18, 48, 50, 65, 86, 88–90, 99, 104, 116

encryption, 36, 42–43, 47, 51–52, 55–58, 136, 140, 144
  server-to-client, 143
  standards, 3
environment, multitenant, 106
errors
  fatal, 74, 125
  handling, 73, 77
  log, 116
  silence, 73, 78
escapeshellarg, 28, 105
escapeshellcmd, 105
eval, 8, 79, 98, 178

F

fail2ban, 113–14, 142
  configured, 114
Ferrara, Anthony, 45
form
  processing, 160
  submissions, 42, 153–55, 159, 167, 169
fuzz, 19, 138

G

garbage collection time, 146
GitHub, 53, 109, 145, 173–74
GitLab, 53
Google, 107, 145
Grossman, Josh, 13

H

hash
  algorithms, 44, 100, 136
  collisions, 100
  cryptographic, 44, 51
  password, 38, 44
  secure, 140
  tables, 99
HashiCorp Vault, 55
HeartBleed, 104, 109, 185
HHVM, 109
HMAC, 36–37, 43, 159, 168–69
  signatures, 36–37
Hornby, Taylor, 52
Hunt, Troy, 181

I

injection attacks, 17, 19, 22, 31, 109, 139
input
  injected SQL, 22
  parses XML, 59
  random, 19
  sanitizing function, 172
  unsanitized, 27, 145
  user-defined, 113
  validating, 135, 138
Insecure Database Lookups, 38, 45
Intrusion detection systems, 128–30, 133

J

JavaScript, 8, 36, 85, 135, 152
- in-page, 152
- unescaped, 83

JavaScript Object Signing and Encryption. See JOSE

joind.in, 181

Joomla, 104

JOSE, 3–4, 36–37, 42
- libraries, 36
- specification, 42

JSON, 3, 95–96, 98–101, 135, 152, 180
- document, 99–100
- Object Signing, 3
- Web Token Libraries, 4, 37
- Web Tokens, 4, 36

K

Krebs, 181

L

Laravel, 178

letsencrypt.org, 77, 143

libsodium, 57

libxml, 61–62

logs, 28, 64, 78, 101, 111–17, 138–39
- server error, 125

M

MAC (message authentication code), 53, 58

md5, 137, 177

Memcached, 41, 124, 140

message authentication code. See MAC

Microsoft Azure, 79

MIME type filtering, 24

ModSecurity, 130

monolog, 116

MySQL, 27, 38–39, 45, 49, 75–76, 80, 87, 90, 108, 124
- truncation error, 90
- Workbench, 80

N

National Institute of Standards and Technology (NIST), 180

National Vulnerability Database, 180

NGINX, beacons, 104

Nomad PHP, 182

nonce, 157–61

O

OAuth, 56, 145
- flow, 144
- token, 63

object injection vulnerabilities, 96–97

OpenID, 59
- Connect, 36, 43, 56

OpenSSL, 104, 109

Open Web Application Security Project[1]. See OWASP

OWASP, 11–14, 122, 130
- mailing list, 12

P

Packagist, 109
Paragon Initiative Enterprises, 179
password, 33, 37–41, 43–46, 48, 55, 63–66, 75, 81, 88, 112, 136–38, 140, 143–44, 184
hashing, 38, 43, 57–58
plaintext, 43–44
strength, 37
PBKDF, 140
PCI, 6, 50
compliance, 49
PHP-based Intrusion Detection System, 128
PHP CodeSniffer
definitions, 178
standards, 178
PHP-FPM, 73
running, 74
PHPMailer, 22, 77, 104–6, 109
PHProxy, 106–7
POST
arguments, 83
body, 152
superglobal, 17
variable, 84
PSR-1, 178
PSR-2, 178
PSR-3 Logger interface, 116

R

RASP, 130, 133
Red Teaming, 174–75, 183
remote code execution attacks, 95
Remote Procedure Calls (RPC), 59, 126, 135
request
frequency, 124
IP, 113
monitoring, 137
throttling, 5, 140
REST interface, 78, 156
Rogue Wave Software, 179, 182
RSA, 36–37, 43
Runtime Application Self-Protection, 130

S

salt, random, 43–45, 140
sanitize, 21, 26–29, 31, 65–66, 77, 88–90, 104, 145, 172
user input, 31, 85, 88
values, 22
Satis, 109
Schneier, Bruce, 181
scripts
arbitrary, 18
client-side, 42, 145
embed, 150
injected, 89
loading, 74
malicious, 138, 156
remote, 30, 74

security
  advisories, 181
  audits, 179
  checklist, 14
  community, 69, 180
security risks, 2, 79, 105, 107, 111, 137
  common, 184
  disclosing, 69
Sendmail, 22, 77, 79, 104–5
Sensio's Security Advisories Checker, 106
Sensitive Data Exposure, 12, 47–48, 50, 52, 54, 56, 58, 122
serialize, 96–97
  native, 95
server
  certificate, 60
  environment, 31, 106
  headers, 104
  multiple, 40, 49
  staging, 54, 72
  tokens, 73, 76
ServerName, 77
Server-Side Request Forgery (SSRF), 59
session
  active, 64–65, 67
  client-side, 35, 40
  cookies, 42, 139
  encrypted, 41
  expired, 21, 26–27
  handler, 40
  ID, 20–21, 25, 35–36, 40
  identifier, 35–36, 139, 146, 151
  server-side, 40, 64
  stores, 34–35
  takeover, 145
  timeouts, 145
  tokens, 33, 159
Sessionz, 41
Shellshock, 109
Slack, 116
Slim, 140, 157–58
  CSRF, 157
SOAP, 59, 135, 156
  interface, 139
sodium, 57–58, 140
SQL, 17, 22, 178
  arbitrary, 172
  injection, 20, 25, 26m 31, 86
  parameterized statement, 25
SSH
  key, 60
  tunnel, 80
SSL certificates, 73, 77, 166
SSRF (Server-Side Request Forgery), 59
Static Code Analysis, 177
Stored XSS, 85, 87, 89
Stripe, 50
stripslashes, 20
Symfony, 68–69, 156, 158, 161, 178
  Authorization, 68
  CSRF component, 161
  Security Component, 68

T

timing attack, 45
TimThumb, 23–24, 30–31
tokens
  access, 3, 138
  authentication, 145
  custom, 144
  non-random, 154
  reset, 39, 45
  signing, 36
Transport layer security, 144
Travis CI, 54
Twitter, 166, 173, 180
two-factor authentication code, 63

U

unserialize, 35, 95–96, 98, 100
users
  authenticated, 14, 63, 67–69, 113–15, 153
  privileged, 19, 54
UUID, random, 136

V

validation, 4, 58, 88, 159–60, 167, 169
VPN, 50–51
vulnerability disclosure, 122, 181, 185

W

WAF (web application firewall), 13, 129–30, 132–33
  dynamic, 129
  open-source, 130
web application firewall. See WAF
whitelist, explicit, 23, 27, 100, 167
WordPress, 22, 26, 68–69, 87, 90, 104, 114, 126–28, 130–32, 136–37, 143, 157–60, 178, 180, 184
  filters, 29
  nonces, 159
  plugin, 20, 39, 114, 131
  SQL injection, 25
  team, 184
  XML-RPC interface, 131, 137

X

XEE. See XML External Entities
xkcd.com, 19, 37
XML, 59, 61–62, 95, 135
  External Entities (XEE), 12, 59–60, 62, 122
  parser, 62
XML-RPC, 126, 131–33, 136–37, 139, 156
  code, 132
XSS, 12, 83–84, 86–88, 90, 122
  attack, 83–84, 87, 150
  reflected, 84, 88
XXE, 59–60, 62

Y

Yubikeys, 55

Z

Zend, 182
ZendCon, 182
Zend Framework, 178