PHP Magic Tricks: Type Juggling

Who Am I

Chris Smith (@chrismsnz)

Previously:

- Polyglot Developer - Python, PHP, Go + more
- Linux Sysadmin

Currently:

- Pentester, Consultant at Insomnia Security
- Little bit of research

Insomnia Security Group Limited

Founded in 2007 by Brett Moore.

New Zealand-based company.

Offices in Auckland and Wellington, as well as global partners.

Brings together a team of like-minded, highly technically skilled, results-driven, security professionals.

CREST Certified Testers.

Regularly perform work for customers in such differing industries as:

- Tele- and Mobile Communications;
- Banking, Finance, and Card Payment;
- E-Commerce and Online Retail;
- Software and Hardware Vendors;
- Broadcasting and Media; and
- Local and National Government.

Conventions

Types:

- `"string"` for strings
- `int(0), float(0)` for numbers
- `TRUE, FALSE` for booleans

Terms:

- "Zero-like" - an expression that PHP will loosely compare to `int(0)`

What is Type Juggling?

Present in other languages, but in PHP, specifically:

- Has two main comparison modes, lets call them loose (==) and strict (===).
- Loose comparisons have a set of operand conversion rules to make it easier for developers.
- Some of these are a bit weird.

PHP Comparisons: Strict

Strict comparisons with ===

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"1"	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
array()	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
"php"	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

PHP Comparisons: Loose

Loose comparisons with ==

	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE
1	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
0	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
-1	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
"1"	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
"0"	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
"-1"	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
NULL	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
array()	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
"php"	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
""	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE

PHP Comparisons: Loose

When comparing a string to a number, PHP will attempt to convert the string to a number then perform a numeric comparison

- `TRUE: "0000" == int(0)`
- `TRUE: "0e12" == int(0)`
- `TRUE: "1abc" == int(1)`
- `TRUE: "0abc" == int(0)`
- `TRUE: "abc"  == int(0) // !!`

PHP Comparisons: Loose

It gets weirder... If PHP decides that both operands look like numbers, even if they are actually strings, it will convert them both and perform a numeric comparison:

- `TRUE: "0e12345" == "0e54321"`
- `TRUE: "0e12345" <= "1"`
- `TRUE: "0e12345" == "0"`
- `TRUE: "0xF" == "15"`

Less impact, but still important.

PHP Type Juggling Bugs

Very common, as == is the default comparison in other languages

Difficult to actually exploit, due to usually not being able to input typed data via HTTP, only strings

Usually manifest as bugs in hardening or protections, allowing you to exploit other bugs that would otherwise be mitigated

Bug #1 - Laravel CSRF Protection Bypass

I discovered this bug November 2014

Was looking around at different places Type Juggling bugs could affect application security.

Bug was very easy to find - first place I looked

A bit harder to exploit

Bug #1: The Bug

```
if (Session::token() != Input::get('_token'))
{
      throw new Illuminate\Session\TokenMismatchException;
}
```

`Session::token()` is the CSRF token retrieved from the session

`Input::get('_token')` is a facade that corresponds to HTTP request input ... sometimes

Bug #1: The Trick

- CSRF token is a `"RaNdOmStRiNg123"`
- What type of juggling can take place here?
- What if: If the CSRF token starts with a letter, or the number 0 (~85% chance)?
- Comparing it with an integer means that PHP will juggle the CSRF token to `int(0)`

Bug #1: The Exploit

Cool story, but how can we make `Input::get('_token')` return `int(0)`?

HTTP Parameters are always strings, never other types

JSON?

Yep. Laravel feeds any request with `'/json'` in the `Content-Type` header through a JSON parser and shoves the result into the `Input` facade

Bug #1: The Exploit

```
$.ajax("http://<laravel app>/sensitiveaction", {
    type: 'post',
    contentType: 'application/x-www-form-urlencoded; charset=UTF-8; /json',
    data: '{"sensitiveparam": "sensitive", "_token": 0}',
});
```

The content type doesn't trigger CORS restrictions (Firefox 34, Chrome 39) but does trigger Laravel JSON parsing

`_token` parameter passes the CSRF check, most of the time

Bug #1: The Aftermath

- Untested, but using `TRUE` as token value should pass 100%
- Reported to Laravel, promptly fixed
- However, the bug did not exist in the framework (which could be patched by composer in a Laravel point release)
- Rather, it was in project template code used to bootstrap new projects - everyone who used the default CSRF protection had to manually apply the patch to their project!
- JSON bug/weakness still stands (Laravel 4)

Bug #2: Laravel Cryptographic MAC Bypass

Laravel again!

Discovered and published by MWR Information Security, June 2013

Bug was in cryptographic library used throughout the framework

The library powered Laravel's authentication system and exposed for use by any Laravel applications

Bug #2: The Bug

A Laravel "encryption payload" looks like this:

```
{
  "iv":    "137f87545d8d2f994c65a6f336507747",
  "value": "c30fbe54e025b2a509db7a1fc174783c35d023199f9a0e24ae23a934277aec66"
  "mac":   "68f6611d14aa021a80c3fc09c638de6de12910486c0c82703315b5d83b8229bb",
}
```

The MAC check code looked like this:

```
$payload = json_decode(base64_decode($payload), true);

if ($payload['mac'] != hash_hmac('sha256', $payload['value'], $this->key))
    throw new DecryptException("MAC for payload is invalid.");
```

Bug #2: The Trick

The calculated MAC (i.e. the result of `hash_hmac()`) is a string containing hexadecimal characters

The use of a loose comparison means that if an integer was provided in the JSON payload, the HMAC string will be juggled to a number

- `"7a5c2...72c933" == int(7)`
- `"68f66...8229bb" == int(68)`
- `"092d1...c410a9" == int(92)`

Bug #2: The Exploit

If the calculated MAC is "68f66...8229bb" then the following payload will pass the MAC check:

```
{
  "iv":    "137f87545d8d2f994c65a6f336507747",
  "value": "c30fbe54e025b2a509db7a1fc174783c35d023199f9a0e24ae23a934277aec66"
  "mac":   68,
}
```

Now you can alter the ciphertext, `"value"`, to whatever you please, then repeat the request until a matching MAC input is found

Bug #2: The Aftermath

The MAC bug allows an attacker to submit arbitrary ciphertexts and IV's which are processed by the server in CBC mode

Arbitrary ciphertexts + CBC + poor error handling = Padding Oracle!

With a Padding Oracle, you can:

- Decrypt any encrypted ciphertexts
- Forge valid ciphertexts for arbitrary plaintexts

Without knowing the underlying encryption key

Bug #2: The Aftermath

Laravel's encryption library powered its "Remember Me" authentication functionality

```
s:4:"1337"; + <padding>
```

This juggling bug allowed exploitation of the crypto flaws, leading to:

- Impersonation of any application user via. Remember Me cookie
- Remote Code Execution by leveraging PHP serialisation bugs:
  - Magic Method execution of existing classes
  - Other bugs (including recent DateTime Use After Free RCE)

Bug #3: Wordpress Authentication Bypass

Publicised by MWR Information Security (again) November 2014

Fun and interesting attack, but limited practicality

Probably easier ways to own Wordpress

Following is a simplified explanation of the bug

Bug #3: The Bug

```
$hash = hash_hmac('md5', $username . '|' . $expiration, $key);

if ($hmac != $hash) {
    // bad cookie
}
```

`$username`, `$expiration` and `$hmac` are provided by the user in the cookie value

`$key` for all intents and purposes is secret

Bug #3: The Trick

The calculated hash, the result of `hash_hmac()`, looks like:

```
"596440eae1a63306035942fe604ed854"
```

The provided hash, given by the user in their cookie, may be any string

If we can make the calculated hash string Zero-like, and provide `"0"` in the cookie, the check will pass

```
"0e768261251903820937390661668547" == "0"
```

Bug #3: The Exploit

You have control over 3 elements in the cookie:

- `$username` - username you are targetting, probably `"admin"`
- `$hmac` - the provided hash, `"0"`
- `$expiration` - a UNIX timestamp, must be in the future

```
hash_hmac(admin|1424869663) -> "e716865d1953e310498068ee39922f49"
hash_hmac(admin|1424869664) -> "8c9a492d316efb5e358ceefe3829bde4"
hash_hmac(admin|1424869665) -> "9f7cdbe744fc2dae1202431c7c66334b"
hash_hmac(admin|1424869666) -> "105c0abe89825a14c471d4f0c1cc20ab"
```

Bug #3: The Exploit

Increment the expiration timestamp enough times and you will eventually get a Zero-like calculated HMAC:

```
hash_hmac(admin|1835970773) -> "0e174892301580325162390102935332"
```

Which makes the comparison:

```
"0e174892301580325162390102935332" == "0"
```

Enough times = 300,000,000 requests avg, ~30 days @ 100 req/s

Bug #3: The Aftermath

Can (eventually) impersonate any user of the Wordpress installation

Code has since been updated:

- `SHA1/256` instead of `MD5`, much harder to get a Zero-like hash
- Updated to use `hash_equals()` instead of `==`, constant time, type safe
- Also now includes another unique token

Recap

PHP's Type Juggling magic trick, a developer convenience, has unexpected behaviour that might bite you

Difficult to exploit, as HTTP Request parameters are usually always strings, but even then you can cause PHP to juggle

Security-sensitive developers need to know how PHP acts in these situations, unpredictability can be catastrophic

Recommendations

Use === as your default comparison. Only reach for == if you really need it

If you need to convert types, perform explicit type conversions using a cast

```
(int)"0e23812" === (int)"0e48394832"
```

Be very mindful of these issues when writing security-sensitive code

www.insomniasec.com

Chris Smith - @chrismsnz

For sales enquiries: sales@insomniasec.com
All other enquiries: enquiries@insomniasec.com
Auckland office: +64 (0)9 972 3432
Wellington office: +64 (0)4 974 6654

References

CSRF Vulnerability in Laravel 4

http://blog.laravel.com/csrf-vulnerability-in-laravel-4/

Laravel Cookie Forgery, Decryption and RCE

https://labs.mwrinfosecurity.com/blog/2014/04/11/laravel-cookie-forgery-decryption-and-rce/

Wordpress Auth Cookie Forgery

https://labs.mwrinfosecurity.com/blog/2014/04/11/wordpress-auth-cookie-forgery/

Writing Exploits for Exotic Bug Classes: PHP Type Juggling

https://www.alertlogic.com/blog/writing-exploits-for-exotic-bug-classes-php-type-juggling/

PHP Documentation: Type Juggling

http://php.net/manual/en/language.types.type-juggling.php

BONUS BUG

Lets take strcmp():

```
int strcmp(string $str1, string $str2)
```

- Returns -1 if $str1 < $str2
- Returns 0 if $str1 === $str2
- Returns +1 if $str1 > $str2

BONUS BUG

How would you use this function?

```
if (strcmp($_POST['password'], 'thePassword') == 0) {
  // do authenticated things
}
```

You control `$_POST['password']`, can you do anything to disrupt this check?

BONUS BUG

Instead of POSTING a password string:

```
password=notThePassword
```

Submit an array:

```
password[]=
```

PHP translates POST variables like this to an empty array which causes strcmp() to barf:

```
strcmp(array(), "thePassword") -> NULL
```

BONUS BUG

Lets take a look at the strcmp usage again:

```
if (strcmp($_POST['password'], 'thePassword') == 0) {
  // do authenticated things
}
```

Lucky for us, thanks to type juggling, NULL == 0. Auth bypass!