Jarosław "msm" Jedynak, Mateusz "Rev" Szymaniec

CONFidence CTF 2015 – PHP Core

CONFidence CTF 2015 to już druga edycja offline'owych zawodów typu Capture The Flag organizowanych przez polski zespół Dragon Sector. Zadania przygotowane przez mistrzów świata (wg rankingu ctftime.org) były wymagające, a niektóre z nich pozostały nierozwiązane mimo obecności nawet najbardziej doświadczonych drużyn. Warto wspomnieć, że w tegorocznej edycji po raz pierwszy pojawiły się nagrody pieniężne, o które pod koniec maja do Krakowa na konferencję Confidence przyjechało walczyć kilkanaście zespołów z całej Europy.

CTF	CONFidence CTF 2015 (Finał) https://ctf.dragonsector.pl
Waga CTFtime.org	30 (https://ctftime.org/event/206)
Liczba drużyn (z niezerową liczbą punktów)	11
System punktacji zadań	Od 100 punktów (proste) do 500 punktów (trudne)
Liczba zadań	20
Podium	1. More Smoked Leet Chicken (Rosja) – 3000 pkt. 2. H4x0rPsch0rr (Niemcy) – 2900 pkt. 3. !SpamAndHex (Węgry) – 2700 pkt.
Zadanie	PHP Core (Reverse 300)

PHP CORE

Zadanie „PHP Core" z kategorii *reverse* było ocenione na 300 punktów, rozwiązało je 7 zespołów, a jego autor to Gynvael Coldwind. Zachęcamy do spróbowania rozwiązania zadania we własnym zakresie, a potrzebne dane znajdziecie pod adresem: http://gynvael.coldwind.pl/download.php?f=php_core_task.zip.

Razem z zadaniem otrzymujemy dwa pliki, *php.core* oraz *php*. Jak wskazują na to ich nazwy (oraz dla pewności program *file*), pierwszy z nich to core dump powiązany z procesem interpretera PHP, którego plik wykonywalny to ten drugi plik.

Core dump to zrzut pamięci danego procesu do pliku, najczęściej tworzony (automatycznie przez system operacyjny) w momencie jego nieoczekiwanego zakończenia. Pomaga on w ustaleniu przyczyny wystąpienia błędu w procesie zwanym debugowaniem pośmiertnym (post-mortem). W Linuksach core dump jest najczęściej plikiem typu ELF, w którym jako segmenty zawarte są surowe dane z kolejnych stron pamięci oraz dodatkowe informacje, w tym stan rejestrów procesora. Plik ten możemy załadować do GDB, który na podstawie metadanych w połączeniu z dostępnymi symbolami debugowania pozwoli na analizę ramek stosu, włącznie z wartościami argumentów czy zmiennych globalnych.

WSTĘPNA ANALIZA PLIKU

Na nasze szczęście autor zadania dołączył również plik wykonywalny, z którego powstał core dump. Miał on również symbole debugowania. Nie pozostało nam w takim razie nic innego jak wczytanie ich do GDB.

```
# gdb php php.core
(...)
Core was generated by `/home/secret/php-5.6.8/out/bin/php
crackme.php'.
Program terminated with signal SIGTRAP, Trace/breakpoint trap.
#0  zend_execute (op_array=0x7ffff7fc7188) at /home/secret/
php-5.6.8/Zend/zend_vm_execute.h:385
```

Na podstawie tych informacji możemy powiedzieć, że autor zadania wygenerował core dump tuż przed wywołaniem funkcji `zend_execute`, na której ustawił breakpoint.

```
(gdb) backtrace
#0  zend_execute (op_array=0x7ffff7fc7188) at /home/secret/
php-5.6.8/Zend/zend_vm_execute.h:385
#1  0x000000000061bf35 in zend_execute_scripts (type=8,
retval=0x0, file_count=3) at /home/secret/php-5.6.8/Zend/
zend.c:1341
#2  0x000000000058ecd9 in php_execute_script (primary_
file=0x7fffffffd2d0) at /home/secret/php-5.6.8/main/main.c:2597
#3  0x00000000006c791d in do_cli (argc=2, argv=0xa5e940) at /
home/secret/php-5.6.8/sapi/cli/php_cli.c:994
#4  0x00000000006c8af4 in main (argc=2, argv=0xa5e940) at /home/
secret/php-5.6.8/sapi/cli/php_cli.c:1378
```

Z powyższego ciągu wywołań funkcji możemy domyślić się, że najprawdopodobniej działanie programu zostało zatrzymane przed wywołaniem skryptu PHP i jednym z celów zadania będzie poznanie jego działania. Na tym etapie ciekawym jeszcze wydaje się argument `primary_file` z funkcji `php_execute_script`:

```
(gdb) frame 2
#0  0x000000000058ecd9 in php_execute_script (primary_
file=0x7fffffffd2d0) at /home/secret/php-5.6.8/main/main.c:2597
2597    /home/secret/php-5.6.8/main/main.c: No such file or
directory.
(gdb) p *primary_file
$1 = {type = ZEND_HANDLE_MAPPED, filename = 0xa5e990 "crackme.
php", opened_path = 0x0, handle = {fd = -134450432, fp =
0x7ffff7fc7300,
  stream = {handle = 0x7ffff7fc7300, isatty = 0, mmap = {len =
  1997, pos = 0, map = 0x7ffff7ff4000,
  buf = 0x7ffff7ff4000 <error: Cannot access memory at address
  0x7ffff7ff4000>, old_handle = 0xb48ec0,
  old_closer = 0x6399f4 <zend_stream_stdio_closer>}, reader =
  0x6399c5 <zend_stream_stdio_reader>,
  fsizer = 0x639a22 <zend_stream_stdio_fsizer>, closer = 0x639b32
  <zend_stream_mmap_closer>}}, free_filename = 0 '\000'}
```

Są to informacje o samym pliku ze skryptem przekazanym interpreterowi, ale niestety sam bufor z treścią pliku (albo jego częścią) jest już niedostępny.

W takim razie prawdopodobnie najciekawszą funkcją do analizy będzie ta bieżąca, czyli `zend_execute`. Na podstawie źródeł PHP dowiadujemy się, że to ta funkcja odpowiada za faktyczne wywołanie skryptu PHP, o którym przyjmuje informacje w argumencie `op_array`. Wybierzmy odpowiednią ramkę i zbadajmy ten argument:

```
(gdb) frame 0
#2  zend_execute (op_array=0x7ffff7fc7188) at /home/secret/
php-5.6.8/Zend/zend_vm_execute.h:385
385     in /home/secret/php-5.6.8/Zend/zend_vm_execute.h
(gdb) p op_array
$3 = (zend_op_array *) 0x7ffff7fc7188
(gdb) p *op_array
$4 = {type = 2 '\002', function_name = 0x0, scope = 0x0, fn_flags
= 134217728, prototype = 0x0, num_args = 0, required_num_args =
0,
  arg_info = 0x0, refcount = 0x7ffff7fc7580, opcodes =
  0x7ffff7fc75d8, last = 50, vars = 0x7ffff7fc8b90, last_var = 7,
  T = 24,
  nested_calls = 1, used_stack = 4, brk_cont_array =
  0x7ffff7fc6378, last_brk_cont = 2, try_catch_array = 0x0, last_
  try_catch = 0,
  has_finally_block = 0 '\000', static_variables = 0x0, this_var =
  4294967295, filename = 0x7ffff7fc73c0 "/home/secret/crackme.php",
  line_start = 0, line_end = 0, doc_comment = 0x0, doc_comment_
  len = 0, early_binding = 4294967295, literals = 0x7ffff7fc94e8,
  last_literal = 23, run_time_cache = 0x0, last_cache_slot = 7,
  reserved = {0x0, 0x0, 0x0, 0x0}}
```

BYTECODE PHP I JEGO DEASEMBLACJA

Po wczytaniu skryptu w formie tekstowej, interpreter PHP każdą z funkcji kompiluje do bytecode'u, czyli ciągu opcode'ów: pojedynczych i prostych do wykonania instrukcji. Każda z nich obsługiwana jest przez *handler*, czyli funkcję po stronie silnika PHP odpowiadającą za jej faktyczne działanie. Instrukcje pobierają maksymalnie dwa operandy (argumenty) oraz opcjonalnie zwracają wynik, a mogą być one zmiennymi bądź wartościami tymczasowymi (na wzór rejestrów). Używany jest również stos do przekazania argumentów funkcjom PHP. Wszystkie opcode'y opisane są krótko na stronach podręcznika PHP (http://php.net/manual/en/internals2.opcodes.list.php), ale ze względu na często zbyt lakoniczne opisy musieliśmy posiłkować się samymi źródłami (https://github.com/php/php-src/blob/master/Zend/zend_vm_def.h).

Z przekazanej do funkcji struktury dowiadujemy się wielu ciekawych informacji: cała funkcja zawiera w sobie 7 zmiennych, 23 literały i składa się z 50 opcode'ów. Zagłębiając się w tę strukturę, możemy bliżej poznać również wszystkie te elementy.

Lista wszystkich zmiennych i literałów, które używane są przy wywołaniu funkcji, znajduje się odpowiednio w tablicach `vars` oraz `literals`. Wewnętrznie opcode'y używają tylko indeksów (lub offsetów w przypadku wartości tymczasowych), a na podstawie tych danych poznamy nazwy zmiennych oraz wartości literałów:

```
(gdb) p op_array.vars[0]
$5 = {name = 0x7ffff7ea77f0 "a", name_len = 1, hash_value =
5863110}

(gdb) p op_array.literals[0]
$6 = {constant = {value = {lval = 140737352726416, dval =
6,9533491068764041e-310, str = {val = 0x7ffff7ea7790 "Please
enter password: ",
    len = 23}, ht = 0x7ffff7ea7790, obj = {handle = 4159338384,
  handlers = 0x17}, ast = 0x7ffff7ea7790}, refcount__gc = 2,
  type = 6 '\006', is_ref__gc = 1 '\001'}, hash_value = 0, cache_
  slot = 4294967295}

(gdb) p op_array.opcodes[0]
$7 = {handler = 0x66263d <ZEND_ECHO_SPEC_CONST_HANDLER>, op1 =
{constant = 4160525544, var = 4160525544, num = 4160525544,
  hash = 140737353913576, opline_num = 4160525544, jmp_addr =
  0x7ffff7fc94e8, zv = 0x7ffff7fc94e8, literal = 0x7ffff7fc94e8,
  ptr = 0x7ffff7fc94e8}, op2 = {constant = 0, var = 0, num = 0,
  hash = 0, opline_num = 0, jmp_addr = 0x0, zv = 0x0, literal =
  0x0,
  ptr = 0x0}, result = {constant = 0, var = 0, num = 0, hash = 0,
  opline_num = 0, jmp_addr = 0x0, zv = 0x0, literal = 0x0, ptr =
  0x0},
  extended_value = 0, lineno = 8, opcode = 40 '(', op1_type = 1
'\001', op2_type = 8 '\b', result_type = 8 '\b'}
```

Z wartości w strukturze opcode'u możemy ją zidentyfikować (po numerze w polu `opcode`), poznać handler oraz rodzaj argumentów oraz wartości zwracanej. Część opcode'ów obsługuje wiele typów argumentów i na podstawie pól `op1_type`, `op2_type` oraz `result_type` dowiemy się, w jaki sposób je interpretować: 1 - stała, 2 - wartość tymczasowa, 4 - zmienna, 8 - brak faktycznego argumentu, 16 - zoptymalizowana wartość tymczasowa („compiled variable").

W trakcie rozpoznania udało nam się dotrzeć do deasemblera bytecode'u PHP (VLD, http://pecl.php.net/package/vld), który na podstawie powyższych struktur potrafi odtworzyć jego tekstową reprezentację. Niestety da się go uruchomić tylko w trakcie działania interpretera jako moduł PHP. Oceniliśmy, że zmodyfikowanie aplikacji do potrzeb analizy podczas debugowania post-mortem zajęłoby nam więcej czasu niż ręczne przeanalizowanie wszystkich 50 opcode'ów.

W końcu, po długiej walce z gdb, podręcznikiem PHP oraz źródłami interpretera dotarliśmy do takiej postaci naszej funkcji:

```
0:  ECHO            "Please enter password: ",
1:  FETCH_CONSTANT  STDIN
2:  SEND_VAL        [ST]
3:  SEND_VAL        1024
4:  DO_FCALL        fread
```

```
 5:  ASSIGN          $a
 6:  SEND_VAR        $a
 7:  DO_FCALL        trim
 8:  ASSIGN          $a, [TMP]
 9:  SEND_VAR        $a
10: DO_FCALL         strlen
11: IS_NOT_IDENTICAL [TMP], 51
12: JMPZ             15
13: EXIT
14: JMP              15
15: ASSIGN           $x, ""
16: ASSIGN           $i, 8
17: SEND_VAR         $a
18: DO_FCALL         strlen
19: IS_SMALLER       $i, [TMP]
20: JMPZNZ           24, 43
21: POST_INC         $i
22: FREE
23: JMP              19
24: SEND_VAR         $a
25: SUB              0, $i
26: SEND_VAL         [TMP]
27: SEND_VAL         8
28: DO_FCALL         substr
29: ASSIGN           $k, [TMP]
30: SEND_VAR         $k
31: DO_FCALL         md5
32: ASSIGN           $l, [TMP]
33: ASSIGN           $j, 31
34: IS_SMALLER_OR_EQUAL $j, 0
35: JMPZNZ           39, 42
36: POST_DEC         $j
37: FREE
38: JMP              34
39: FETCH_DIM_R      $l, $j
40: ASSIGN_CONCAT    $x, [TMP]
41: JMP              36
42: JMP              21
43: ASSIGN           $w, <długi string z hashami>
44: IS_IDENTICAL     $x, $w
45: JMPZ             48
46: ECHO             "Access Granted!\n"
47: JMP              49
48: EXIT
49: RETURN
```

Znaczenie większości opcode'ów jest łatwe do domyślenia się z samych ich nazw, ale dla ułatwienia czytania zamieszczamy krótki opis każdej występującej instrukcji:

`ASSIGN a, b`
`ASSIGN_CONCAT a, b`
przypisuje zmiennej a wartość b (która może być stałą, drugą zmienną albo wartością tymczasową, będącą tutaj zawsze wynikiem poprzedniej operacji - oznaczaną przez nas jako [TMP]). Wersja z CONCAT to połączenie przypisania i konkatenacji, czyli operator `.=`

`ECHO a`
bezpośredni odpowiednik `echo` z PHP, wypisuje a na standardowe wyjście

`FETCH_CONSTANT a`
pobiera zdefiniowaną stałą o nazwie podanej w a

`SEND_VAL a`
`SEND_VAR a`
umieszcza podaną w a wartość (wersja VAL) lub zmienną (wersja VAR) na stosie jako argument następnej wykonywanej funkcji

`DO_FCALL a`
wywołuje funkcję o nazwie podanej w a

`IS_NOT_IDENTICAL a, b`
`IS_SMALLER a, b`
`IS_SMALLER_OR_EQUAL a, b`
porównuje dwa argumenty, odpowiednio odpowiedniki operatorów `!=`, `<` oraz `<=`

`ADD a, b`
dodaje a i b, odpowiednik operatora +

`SUB a, b`
odejmuje b od a, odpowiednik operatora -

`FETCH_DIM_R a, b`
pobiera b-ty element tablicy a, odpowiednik `a[b]`

`POST_DEC a`
postdekrementacja zmiennej a, odpowiednik operatora `--`

`FREE`
zdejmuje niepotrzebną wartość ze stosu (używane np. po dekrementacji, której zwracaną wartość chcemy zignorować)

`JMP a`
skok bezwarunkowy, kontynuuje wykonywanie od opcode'u o indeksie a

`JMPZ a`
skok do opcode'u pod indeksem a, jeśli wynik poprzedniej operacji jest równy 0

`JMPZNZ a, b`
skok do opcode'u pod indeksem a, jeśli wynik poprzedniej operacji jest równy 0, bądź pod b w przeciwnym wypadku

`EXIT a`
natychmiastowo kończy wykonywanie skryptu, wypisując a na standardowe wyjście, odpowiednik `die`

DEKOMPILACJA

Najtrudniejsza (i najdłuższa) część już za nami - widać tutaj pewne konstrukcje typowe dla języków wysokopoziomowych (pętle, instrukcje warunkowe), i dość łatwo było odzyskać z tego kod PHP odpowiadający oryginałowi:

```php
echo 'Please enter password:';

$a = trim(fread(STDIN, 1024));

if (strlen($a) != 51) {
  die("Not really.\n");
}
$x = '';
for($i = 8; $i < strlen($a); $i++) {
  $l = md5(substr($a, -$i, 8));
  $j = 31;
  while ($j-- >= 0) {
    $x .= $l[$j];
  }
}

if ($x != '[długi ciąg znaków szesnastkowych]') {
  die("ACCESS DENIED!\n");
}
echo 'Access Granted!\n';
```

Widać tutaj, że program:

- pobiera od użytkownika maksymalnie 1024 znaki;
- sprawdza, czy napis wprowadzony przez użytkownika ma dokładnie 51 znaków (jeśli nie, to kończy działanie);
- bierze ośmioznakowe podciągi podanego hasła (zaczynając od końca), z każdego liczy skrót md5, odwraca go i dokleja do wynikowego napisu $x;
- pod koniec porównuje $x z długą stałą (którą udało nam się odzyskać z pamięci), i w zależności od wyniku porównania akceptuje lub odrzuca hasło.

Inaczej ujmując problem, ktoś policzył md5 dla każdego ośmioznakowego podciągu hasła (tzw. „sliding window"), wszystkie je odwrócił i skleił ze sobą. A my, na podstawie wyniku tej operacji, chcemy poznać wejściowe hasło.

ZDOBYWANIE FLAGI

Rozpoczęliśmy od sprawdzenia (w bazie serwisu hashkiller.co.uk), czy przypadkiem któryś hash nie został już przez kogoś złamany wcześniej:

```
f61e5b1fa9e0116bd1ef6c71e6d47332 [Not found]
7adb4b3d339d3d74eb20b855591086a3 [Not found]
4df5fbf9422f6f54d5ac48816939df14 [Not found]
de4bf9d9854fb91f93967f32870e6e82 [Not found]
f019a7b09f1fe9873dfa15be2bbc9d05 [Not found]
87e53217e9907c3383ce1a26d9421b0d [Not found]
75771142b5962264051b420f4adf070a [Not found]
0b2ce859a23d34d8cb955dff5d687b9e [Not found]
b846a53743f1f60f64a79432480db727 [Not found]
ba91d0143bec412daec80df75e13b7a1 [Not found]
f2127b07b91c59a638c173c8dc365078 [Not found]
50c58575db2cc850c69fea18b3c1a70c [Not found]
2ae994af2e55e3a9f4cb8b5262f6291e [Not found]
d98f1485e4a6412f8ad176e8a5a75dc5 [Not found]
3421571e572b58bf090e0e02e8398eb5 [Not found]
b9ff404195efadf9735b2591428357cc [Not found]
d8052abfaaeb3f5b0c03844e4ca49237 [Not found]
9ff3393b4aeb77ed3e194c5fb27093ad [Not found]
f9ae5e5c44e59f94680af569879f704a MD5 : Everyone
3c50d6bdd78b4aa13df403cfbb115d60 [Not found]
2dfdebc7c5771f7cdb1357081566adad [Not found]
66e46becbcd8f47081387f80bba62b14 [Not found]
05182daf5fa5f461a8c03df47dbed28f [Not found]
1dac5f45156abfbecfb80471aaba628c [Not found]
30bebb0641641193a15230ea35def57b [Not found]
4d42d66f37886c724d84140a70fd6487 [Not found]
9e4e87d98867785760ae51f8a00e09ef [Not found]
208871cf1b6922dbf717e406edceee02 [Not found]
b9576cc7b831b6b2690db2435cc0649e [Not found]
7d43d642974d9476ae61380abc13bd43 [Not found]
0368bc6663fbeb0334ae5ec91ec26805 [Not found]
a71aeed454c570feeb8ddb9417cd62b7 [Not found]
822f9b81131a4a31b2ff9e7435a9f9a1 [Not found]
f88a64bec3ad831c9c53d5f6f554125d [Not found]
71a1a7ed2cb3f79f5a707a1e86cb455c [Not found]
452e24538750791495a8ce73af8b8db5 [Not found]
36f0cb509cc3f2112c115f4e97f3b905 [Not found]
6e8b3177cb0c155f2bf087b8b98ac74d [Not found]
4cd424ea03dc6d10069c519c070d601c [Not found]
9af8e1d2bc5c822cc01e644b152bea94 [Not found]
afbf0897a5a83fdd873dfb032ec695d3 MD5 : Internal
3b0d8875514f18b1f6496a229d900534 [Not found]
0aea41e39a4f5fa3c4ba5afd85889ad8 [Not found]
```

Nie zawiedliśmy się - udało nam się znaleźć dwa fragmenty znane bazom hashy md5. Gdyby to nie wyszło, moglibyśmy zawsze bruteforce'ować pierwszy hash (znamy początek: 'DrgnS{', a sprawdzenie wszystkich możliwości z pozostałymi dwoma znakami zajęłoby ułamek sekundy na współczesnych procesorach).

Jako że podciągi, z których były liczone hashe, nachodziły na siebie, to znając jeden hash, mogliśmy wyliczyć wszystkie inne - wystarczy zacząć, obcinając pierwszy znak z jednej strony, zgadnąć, jaki znak dokleić z drugiej strony (a raczej sprawdzić wszystkie możliwości), i powtarzać tę operację aż do poznania całego hasła.

Pozostało jedynie napisanie skryptu, który wykonałby tę ciężką pracę za nas. Użyliśmy do tego Pythona:

```
hashes = ['F61E5B1FA9E0116BD1EF6C71E6D47332',
'7ADB4B3D339D3D74EB20B855591086A3',
'4DF5FBF9422F6F54D5AC48816939DF14', (...),
'9AF8E1D2BC5C822CC01E644B152BEA94',
'AFBF0897A5A83FDD873DFB032EC695D3',
'3B0D8875514F18B1F6496A229D900534',
'0aea41e39a4f5fa3c4ba5afd85889ad8']

print 'Seek forward'
ndx = hashes.index('F9AE5E5C44E59F94680AF569879F704A')
curr = 'Everyone'
while ndx < len(hashes) - len(curr):
  for i in range(256):
  next = curr[1:] + chr(i)
  if hashlib.md5(next).hexdigest().upper() == hashes[ndx +
  1].upper():
    curr = next
    print curr
    ndx += 1
    break

print 'Seek backward'
ndx = hashes.index('F9AE5E5C44E59F94680AF569879F704A')
curr = 'Everyone'
while ndx > 0:
  for i in range(256):
  next = chr(i) + curr[:-1]
  if hashlib.md5(next).hexdigest().upper() == hashes[ndx
  - 1].upper():
    curr = next
    print curr
    ndx -= 1
    break
```

I w ten sposób zdobyliśmy w końcu upragnioną flagę:
DrgnS{IAmPrettySureEveryoneLovesPHPEngineInternals}

PODSUMOWANIE

Zadanie, które dane nam było tu opisać, nie było trudne, ale jego rozwiązywanie sprawiło nam bardzo dużo przyjemności. W naszej ocenie było to jedno z najciekawszych i najbardziej oryginalnych, z którymi zmierzyliśmy się do tej pory podczas całej naszej przygody z zawodami CTF.

O drużynie

Rozwiązanie zadania PHP Core zostało nadesłane przez **p4**, zespół CTF-owy użytkowników serwisu 4programmers.net mający ambicje na drugie miejsce w Polsce (bo pierwsze już zajęte ;)

https://ctftime.org/team/5152