CS-4235

Introduction to Information Security

Homework Assignments

Wenqi He

School of Computer Science

Georgia Institute of Technology

This collection is organized in reverse chronological order

CS 4235 Project 4 Report

Wenqi He, whe47

May 7, 2019

1 Target 1 Epilogue

1.1

The vulnerability lies in line 18-29 of `/var/payroll/www/account.php`:

```
$expected = 1;
$teststr = $_POST['account'].$_POST['challenge'].$_POST['routing'];
for ($i = 0; $i < strlen($teststr); $i++) {
    $expected = (13337 * $expected + ord($teststr[$i])) % 100000;
}
if ($_POST['response'] != $expected) {
   ...
} else {
    ...
    $db->query(..);
    notify('Changes saved');
}
```

1.2

The PHP server attempts to prevent CSRF by checking the submitted `csrfr` value against the value computed from `account`, `routing` and `csrfc` fields. The vulnerability is that, although the `csrfc` field is supposed to contain the CSRF token, the server never checks its validity, and therefore an attacker who has figured out the verification algorithm can put *any* value in the `csrfc` field, as long as the `csrfr` value is computed correctly based on `csrfc`.

1.3

Since the server is already issuing CSRF tokens, the easiest step to fix the vulnerability is to simply check the submitted token. For example, the conditional statement

```
if ($_POST['response'] != $expected) {
    // Reject
}
```

can be changed to

```
if (
    $_POST['challenge'] != $_SESSION['csrf_token'] or
    $_POST['response'] != $expected
) {
    // Reject
}
```

2 Target 2 Epilogue

2.1

The vulnerability lies in line 32 (and also 6-13) of `/var/payroll/www/index.php`

2.2

- The first vulnerability is in line 6-13 in the handling of POST requests:

  ```
  $action = @$_POST['action'];
  if ($action == 'login') {
      if($_POST['U3B...Z2c'] == 'U3B...Z2c') {
          $auth->login(...);
      }
  } elseif ($action == 'register') {
      $auth->register(...);
  }
  ```

  This is problematic because if control falls through (for example, if the `action` field is omitted), the page still renders, thereby exposing the XSS vulnerability.

- The XSS vulnerability is on line 32:

  ```
  <input type="text" name="login" value="<?php echo @$_POST['login'] ?>">
  ```

  The user submitted '`login`' value is echoed as is into the rendered document. Therefore, if an attacker puts malicious HTML instead of a valid username in the POST request, he/she can alter the content of the document at will, as long as the syntax of the document after injection is still valid.

2.3

The special characters such as `<`, `>` and `"` which have syntactic meanings in HTML must be escaped using HTML encoding. For example, `<` should be replaced with `<`, and `"` with `"`. This can prevent user inputs from being recognized as HTML.

3 Target 3 Epilogue

3.1

The vulnerability lies in line 28-43 of `/var/payroll/www/includes/auth.php`

3.2

The server does have SQLi prevention, but the list of filtered substrings in function `sqli_filter` is not exhaustive. Suppose the `login` value is entered as `username' OR ''='`. Because single quotes are not forbidden, this input will pass the filter. The first SQL statement then becomes

```
SELECT salt FROM users WHERE eid='username' OR ''=''
```

Since `''=''` always evaluates to true, this is still the same as

```
SELECT salt FROM users WHERE eid='username'
```

The second SQL statement, however, becomes

```
... WHERE eid='username' OR ''='' AND password='$hash'
```

Because `AND` has a higher precedence than `OR`, this statement is equivalent to

```
... WHERE eid='username' OR password='$hash'
```

This query will return the desired row even if the password is incorrect, because `eid='username' OR FALSE` is equivalent to `eid='username'`. Thus, by appending `' OR ''='` to a valid username, login can be achieved without knowing the password.

3.3

As a countermeasure to this particular exploit, one can simply add in function `sqli_filter`:

```
$filtered_string = str_replace("'", "", $filtered_string);
```

However, in order to prevent SQL injections in general in PHP, it is better to:

- Use prepared statements provided by libraries such as PDO and MySQLi.
- Check if the given input has the expected data type using built-in input validating functions such as `is_numeric()`, `ctype_digit()`, or Perl compatible Regular Expressions support.
- If numerical input is expected, silently change its type using `settype()` or `sprintf()`.
- Quote each non numeric user supplied value with the database-specific string escape function, such as `mysql_real_escape_string()` and `sqlite_escape_string()`.

CS 4235 Project 3

Wenqi He

March 17, 2019

2

Yes, because the salt is chosen from a small space. If cracking an unsalted password requires worst-case $\mathcal{O}(n)$ time, then a password salted this way would just require $\mathcal{O}(n^2)$ time, which is harder but still feasible. One way to enhance security would be to generate random strings as salts rathers than selecting from commonly used passwords. That way an attackers would need to try all possible combinations of characters, which would be exponentially more difficult. Another possible enhancement is to add a second salt or a pepper.

3

Step I - Factor n into a pair of prime divisors p, q

This was done through a brute-force linear search since there is no known algorithm to compute prime factors directly. Optimization was made based on the fact that

$$\min\{p, q\} \in (1, \lceil n^{1/2} \rceil]$$

and the two primes are unlikely to be too small. The search for p starts from $\lceil n^{1/2} \rceil$ and proceeds down to 2; if p divides n then the factors are simply p and n/p. The process takes less than $\mathcal{O}(n/2)$.

Step II - Compute the private key from p, q, e

Once the prime factors are obtained, the private key can be computed directly. First the Euler's totient function was computed as the modulus:

$$\phi(n) = \phi(pq) = \phi(p)\phi(q) = (p-1)(q-1)$$

The fact that $ed \equiv 1 \pmod{\phi}$ implies that $gcd(e, \phi) = 1$, because otherwise d, the modular multiplicative inverse of e modulo ϕ, cannot exist. Thus, the congruence can be rephrased as the Bézout's identity, where d is the coefficient associated with e:

$$ed + \phi k = gcd(e, \phi) = 1$$

In light of this identity, the private key d was obtained through the extended Euclidean algorithm with e and $\phi(n)$ as inputs.

5

The moduli generated contain common prime factors, which could facilitate factorization. As before, obtaining the private key still involves first factoring $N1$ into p, q and then computing d by applying the extended Euclidean algorithm to e and $(p-1)(q-1)$. However, this time factorization is trivial, because $N2$ shares a common prime factor with $N1$ and computing $gcd(N1, N2)$ amounts to factoring both $N1$ and $N2$.

Step I - Factor $N1$ into a pair of prime divisors p, q using $N2$

$$p = gcd(N1, N2), \quad q = N1/p$$

Step II - Compute the private key from p, q, e

This step is the same as before.

6

The three ciphertexts can be expressed as

$$C_i = M^3 \bmod N_i, \quad i = 1, 2, 3$$

The encrypted message can be recovered using the (generalized) Chinese remainder theorem, which states that given a set of simultaneous congruences

$$x \equiv a_i \pmod{n_i}, \quad 1 < i < r$$

where n_i are pairwise relatively prime, there exists a uniquely determined

$$x \equiv \sum_{i=1}^{r} a_i M_i N_i \pmod{N}$$

where

$$N = \sum_{i=1}^{r} n_i, \quad N_i = \frac{N}{n_i}, \quad M_i N_i \equiv 1 \pmod{n_i}$$

Proof:

$$x \bmod n_k = \Big(\sum_{i=1}^{r} a_i M_i N_i \Big) \bmod n_k = \Big(a_k M_k N_k \Big) \bmod n_k = a_k$$

Step I - Obtain M^3 using the above formula

Step II - Take the cubic root of M^3 to recover the message

This was implemented as a binary search in interval $[0, n]$ instead of using the native `pow()` function because M^3 is too large.

CS 4235 Project 1

Wenqi He, whe47

January 26, 2019

1

a

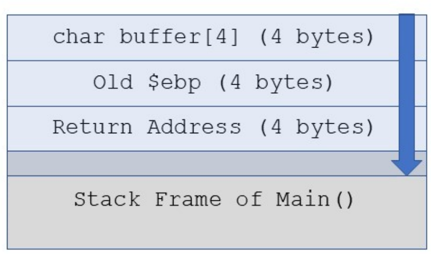
The stack starts from the highest addresses (right before the command-line arguments and environment variables) in the address space and grows towards lower memory addresses. Whenever a function is called, a new stack frame, which contains the arguments passed into the function, current `$eip` (the return address, which points at the next instruction to be executed after the callee returns), current `$ebp` pointing at the calling frame, local variables, etc. (see diagram 1.a), is pushed onto the stack. Local variables are placed right before the location addressed by `$ebp`. Arguments are placed right after the return address. When a function returns, the return address stored on the current stack frame will be used to jump back to the calling function. If the length of a variable can cause misalignment, it would be padded so that it takes up exactly a multiple of the word size.

b

```
#include<stdio.h>

int main(char* args} {
    echo();
}

void echo() {
    char buffer[4];
    gets(buffer);
    printf("%s\n", buffer);
}
```



Function `echo()` read a string from standard input and puts it in a buffer of size 4 bytes on the stack. Typing in a string with more than 4 characters will cause buffer overflow. It takes string of 12 characters to reach and overwrite the return address.

2

a

The heap is located at the lowest memory addresses, right after the text segemnt, initialized data segment and uninitialized data segment. It grows towards higher memory addesses. It starts from the opposite side of the address space and grows in the opposite direction compared to the stack.

b

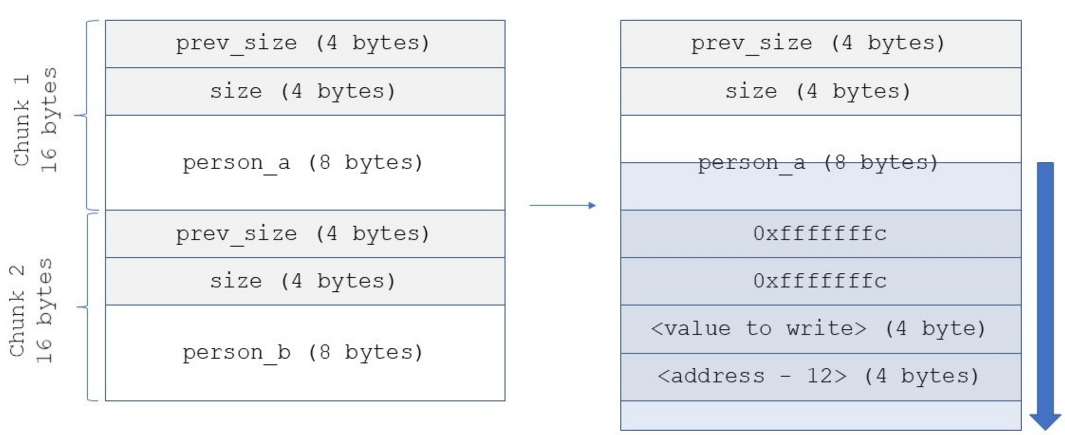
b.

```
#include<stdio.h>
#include<string.h>

typedef struct person {
    void (*greet)(char *name);
    char name[4];
} person;

void greeting_func(char *name) {
    printf("Hi, my name is %s\n", name);
}

int main(char* args} {
    person_a = malloc(sizeof(person));
    person_b = malloc(sizeof(person));
    person_a->greet = greeting_func;
    person_b->greet = greeting_func;
    gets(person_a->name);
    gets(person_b->name);
    person_a->greet(person_a->name);
    person_b->greet(person_b->name);
    free(person_a);
    free(person_b);
}
```



Since the heap is initially empty, `person_b` will be allocated right next to `person_a`. If we provide a name longer than character for `person_a` we can overwrite the metadata of the chunk containing `person_b`. The metadata contains two fields, each of which is 4 bytes long. A function pointer is also 4 bytes long. So if the name is $4 + 4 * 2 + 4 = 16$ bytes long, we can use the last 4 bytes to set the function pointer `person_b->greet` to be any function we want to execute.

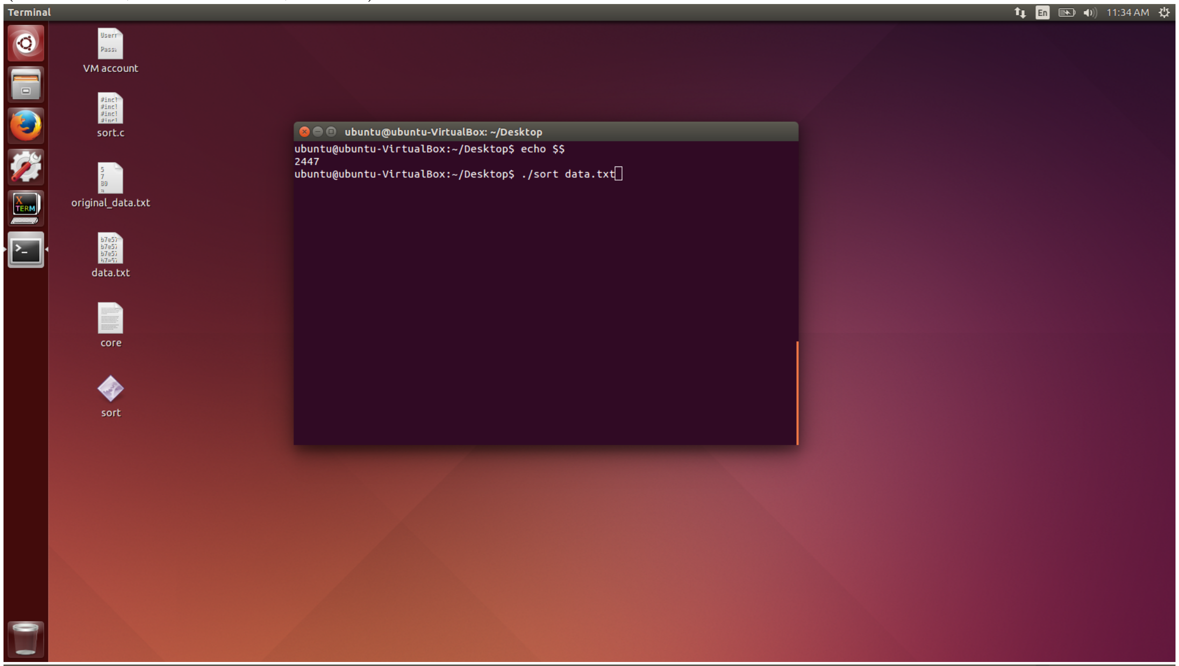
Heap memory is in general not contiguous (except initially). The heap uses doubly linked lists called free-lists to manage unallocated chunks. Each chunk, whether allocated or not, contains the size of the previous chunk and the size of the currrent chunk in its header, and the last bit of the size field indicates whether the previous chunk is allocated. In addition to these two fields, unallocated chunks also contains in its header `FD` and `BK` pointers pointing to adjacent free chunks.

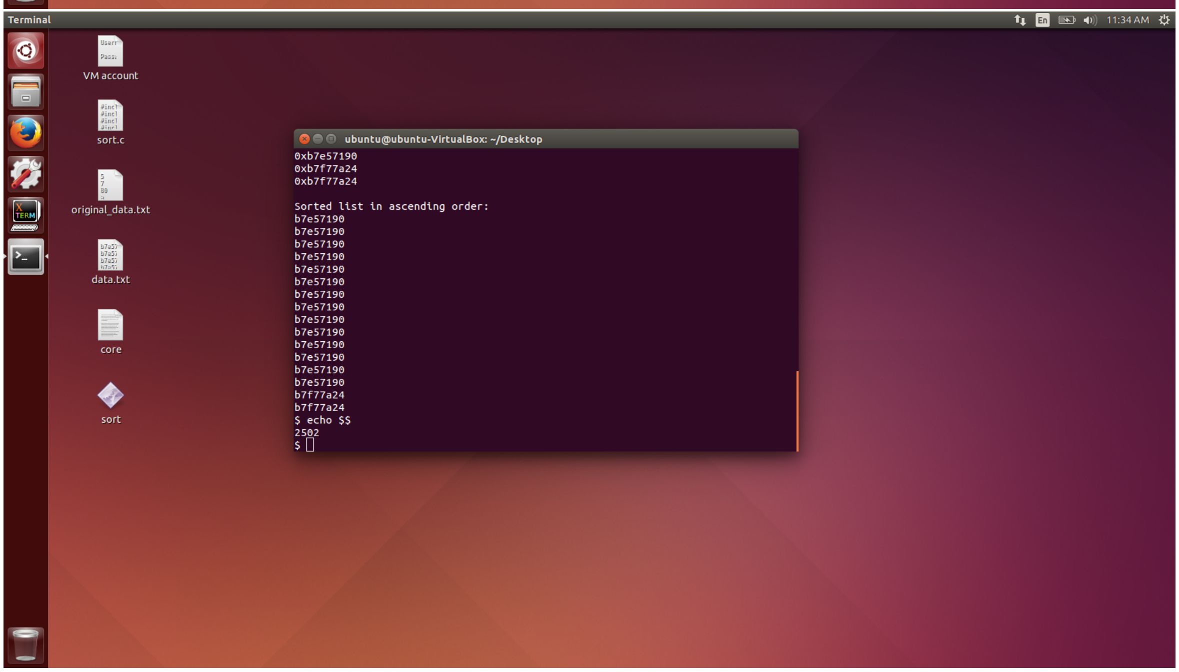
With older implementations of `malloc`, we can exploit the allocator by overwriting heap metadata (chunk headers). The high-level idea is to fake a free chunk to trick the deallocator into coalescing two "free" chunks using the `unlink` macro, which simply does `p->bk->fd = p->fd;` and `p->fd->bk = p->bk;` to remove the chunk from its original free-list. We can use one of those two write operations to overwrite the address of `free` to be the address of some malicious code so that the next time any memory needs to be deallocated, malicious code would run instead.

In the above code example, Suppose `chunk_a` contains `person_a` and `chunk_b` contains `person_b`. When `person_a->name` overflows into `chunk_b`, we can write fake metadata `chunk_b->size = -4`. Now, when `free(person_a)` is called, the deallocator, following its algorithm, will compute the address of the chunk after `chunk_b` to see if a merge is possible, and it will actually get to `chunk_b - 4` and it will interpret `chunk_b - 4 + 8` which is actually `chunk_b -> prev_size` as the `size` field of the "next" chunk. So if we set `chunk_b->prev_size = 0xfffffffc` then the deallocator will see that the last bit is unset, so it would consider `chunk_b` as freed and therefore coalesce `chunk_a` and `chunk_b` and unlink `chunk_b`. We can now utilize the reassignment of linked-list node pointers to write to an arbitrary address. For example, we can put `<func ptr to free> - 12` in `chunk_b->fd` and the address of our injected code in `chunk_b->bk`, which will set the pointer to `free` to point to our code.

3

(Host: PC, Windows 10, 64 bit)





4

Both ROP and JOP are used to circumvent code injection defense mechanisms because they only utilize existing code in the exploited program. Both kinds of attacks are based on manipulating program execution by chaining together snippets of code ending in some control flow instruction (`ret` for ROP, `jmp` for JOP). The main difference between ROP and JOP is that `ret` gadgets can naturally return back the control based on the content of the stack, but `jmp` is uni-directional, so it was previously thought to be difficult for attackers to maintain control. However it has be proven that with an additional dispatcher gadget to govern control flow among various jump-oriented gadgets such attacks are feasible.