بررسی آسیب‌پذیری اجرای کد از راه دور بدون احراز هویت

نرم‌افزار Magento

فهرست مطالب

1 چکیده

مجنتو یک پلتفرم تجارت الکترونیک و فروشگاه ساز متن باز و مبتنی بر PHP است که بیش از ۲۵۰،۰۰۰ فروشگاه آنلاین در سراسر جهان از آن بهره می‌برند و حدود 50 میلیارد دلار در سال به دست می‌آورد. این آمار به همراه این واقعیت که مجنتو تقریبا تمامی اطلاعات مشتریان را ذخیره می‌کند، آن را به یک هدف بسیار حساس تبدیل می‌کند.

یک آسیب‌پذیری اجرای کد از راه دور (Remote code execution) در magento گزارش شده که می‌تواند به مهاجم این امکان را بدهد که بتواند بدون احراز هویت کدهای PHP دلخواه را از طریق APIهای REST یا SOAP روی سیستم قربانی اجرا کرده و به اطلاعات حساس کاربران مانند اطلاعات کارت اعتباری و سایر اطلاعات دست پیدا کند. این موضوع به این دلیل اتفاق می‌افتد که مجنتو نمی‌تواند یک کاربر را برای مقداردهی خطرناک به ویژگی "data_" از نمونه‌ی "payment" از طریق متغیر "method" از تابع "getData()"، به درستی محدود کند. این گزارش به جزییات آن و روش مقابله با خطرات امنیتی مرتبط می‌پردازد.

2 محصولات تحت تاثیر

نسخه‌های قبل از Magento Community Edition 2.0.6 و Enterprise Edition 2.0.6 دارای این آسیب‌پذیری هستند.

3 تاثیر آسیب پذیری

استفاده موفقیت‌آمیز از این آسیب‌پذیری منجر به اجرای کدهای دلخواه توسط مهاجم در زمینه‌ی برنامه‌ی متاثر از این آسیب‌پذیری می‌شود. همچنین استفاده‌ی ناموفق از این آسیب‌پذیری نیز می‌تواند منجر به عدم پذیرش سرویس[1] گردد.

[1] Denial Of Service

4 مشخصه های آسیب پذیری

در این قسمت به مرور کامل آسیب‌پذیری ذکر شده می‌پردازیم.

4–1 خلاصه‌ای از نحوه عملکرد مجنتو

مجنتو سیستمی است که توسط "ماژول"های مختلف ساخته شده است. این به اصطلاح ماژول‌ها، اساسا دایرکتوری های مختلف شامل کدهایی برای ارائه‌ی ویژگی‌های مختلف سیستم هستند. به عنوان مثال، یک ماژول مسئول پرداخت[۲] است، یک ماژول مسئول کارت مجازی[۳] و یکی مسئول مشتریان[۴] است.

در هر ماژول یک دایرکتوری ویژه به نام "API" وجود دارد. این پوشه شامل تمام قابلیت های صادرات همان ماژول به دیگر ماژول‌ها می‌باشد. به این ترتیب ماژول پرداخت می تواند با ماژول سبد خرید، ماژول مشتری با ماژول مجوز، یا ماژول حمل و نقل با ماژول فروش ارتباط برقرار کند.

دایرکتوری API از فایل هایPHP مختلف ساخته شده است که هر یک شامل یک کلاس PHP هستند و مسئولیت اعمال برخی از قابلیت های ماژول را به بقیه سیستم دارند.

ماژول web API در مجنتو اجازه‌ی دو RPC[۵] مختلف REST RPC و SOAP API را می‌دهد. به دلیل اینکه هر دو به طور پیش فرض فعال هستند، ما در این گزارش از SOAP API که از XML بهره می‌برد، استفاده می‌کنیم.

برخی از فراخوانی های API به ما اجازه می دهد اطلاعات خاصی را در افزونه سبد خریدمان[۶] ست کنیم. این اطلاعات می تواند آدرس خرید ما، محصولات و حتی روش پرداخت ما باشد.

4–2 مروری بر آسیب پذیری

آسیب‌پذیری اجرای کد از راه دور یکی از آسیب‌پذیری‌های مهم است که باعث می‌شود نفوذگر با سوء استفاده از این آسیب‌پذیری موجود در نرم‌افزار، دستورات مورد نظر خود را از راه دور بر روی سیستم قربانی اجرا کند.

---

[2] Payment
[3] virtual cart
[4] Customer
[5] Remote Procedure Call
[6] Shopping Cart

3-4 جزییات آسیب پذیری

هنگامی که مجنتو اطلاعات ما را در داخل یک نمونه[7] از سبد خرید قرار می‌دهد، از تابع "save" آن نمونه به منظور ذخیره داده های تازه وارد شده در پایگاه داده استفاده می‌کند. نحوه عملکرد تابع "save" به صورت شکل(1) است.

```
/**
 * Save object data
 */
public function save(\Magento\Framework\Model\AbstractModel $object)
{
    ...
    // If the object is valid and can be saved
    if ($object->isSaveAllowed()) {
        // Serialize whatever fields need serializing
        $this->_serializeFields($object);
        ...
        // If the object already exists in the DB, update it
        if ($this->isObjectNotNew($object)) {
            $this->updateObject($object);
        // Otherwise, create a new record
        } else {
            $this->saveNewObject($object);
        }

        // Unserialize the fields we serialized
        $this->unserializeFields($object);
    }
    ...
    return $this;
}

// AbstractDb::save()
```

شکل(1): عملکرد تابع save() در مجنتو

مجنتو اطمینان حاصل می‌کند که شیء[8] ما معتبر است، هر فیلدی را که باید سریال سازی[9] شود، سریال سازی می‌کند، آن را در پایگاه داده ذخیره می‌کند، و در نهایت فیلدهایی را که قبلا آن را سریال کرده است، unserialize می‌کند.

شکل(2) نشان می‌دهد که چطور مجنتو تصمیم می‌گیرد که کدام فیلد باید سریال سازی شود. همانطور که مشاهده می‌شود، تنها فیلدهای موجود در دیکشنری هاردکدشده‌ی[10] "_serializableFields" می‌توانند سریال سازی شوند و قبل از انجام سریال سازی اطمینان حاصل می‌کند که مقادیر فیلدها یک آرایه یا یک شیء باشند.

---

[7] Instance
[8] Object
[9] Serialize
[10] Hardcoded dictionary

```php
/**
 * Serialize serializable fields of the object
 */
protected function _serializeFields(\Magento\Framework\Model\AbstractModel $object)
{
    // Loops through the '_serializableFields' property
    // (containing hardcoded fields that should be serialized)
    foreach ($this->_serializableFields as $field => $parameters) {
        // Get the field's value
        $value = $object->getData($field);

        // If it's an array or an object, serialize it
        if (is_array($value) || is_object($value)) {
            $object->setData($field, serialize($value));
        }
    }
}

// AbstractDb::_serializeFields()
```

شکل(2): تابع سریال سازی

شکل(3) نشان می‌دهد که چطور مجنتو تصمیم می‌گیرد که کدام فیلد باید unserialize شود. همانطور که مشاهده می‌شود این کد تقریبا همان است. تنها تفاوت این است که این بار Magento اطمینان حاصل می‌کند که مقدار فیلد یک آرایه یا یک شی نیست. با توجه به این دو چک، مهاجم می تواند یک حمله تزریق شی[11] را به سادگی با تنظیم یک رشته عادی به یک فیلد قابل سریال سازی انجام دهد.

```php
/**
 * Unserialize serializeable object fields
 */
public function unserializeFields(\Magento\Framework\Model\AbstractModel $object)
{
    // Loops through the '_serializableFields' property
    // (containing hardcoded fields that should be serialized)
    foreach ($this->_serializableFields as $field => $parameters) {
        // Get the field's value
        $value = $object->getData($field);

        // If it's not an array or an object, unserialize it
        if (!is_array($value) && !is_object($value)) {
            $object->setData($field, unserialize($value));
        }
    }
}

// AbstractDb::unserializeFields ()
```

شکل(3): تابع unserialize

هنگامی که یک رشته عادی را به یک فیلد قابل سریال سازی تبدیل می‌کنیم، سیستم قبل از اینکه شی در پایگاه داده ذخیره کند، آن را سریال نخواهد کرد، زیرا این یک شی یا یک آرایه نیست. اما هنگامی که سیستم سعی می‌کند آن را unserialize کند، درست بعد از اینکه کوئری در پایگاه داده اجرا شد، آن را

[11] object injection attack

unserialize خواهد کرد، زیرا آن یک شی یا یک آرایه نیست. این شرایط کوچک و تقریبا نامرئی، تفاوت بین یک سیستم آسیب‌پذیر و یک سیستم امن را ایجاد می‌کند.

تنها سوالاتی که باقی می‌ماند، این است که کدام فیلدها قابل سریال سازی هستند و چگونه می‌توان آنها را تنظیم کرد. برای سوال اول کافی‌ست جستجو کنیم که کدام کلاس‌ها ویژگی[12] "_serializableFields" را تعریف می‌کنند. چندین کلاس وجود دارد که این ویژگی را تعریف می‌کنند اما هیچ کدام از آنها نمی‌توانند با استفاده از XMLRPC ایجاد شوند. یکی از این کلاسها به نام "Payment" (که مسئول جمع‌آوری اطلاعات در مورد جزئیات پرداخت است)، در یکی از متدهای API ظاهر شده اما نه به عنوان یک آرگومان، بنابراین ویژگی‌های نمونه‌ی آن را نمی‌توان ایجاد یا کنترل کرد. در حقیقت فیلد قابل سریال سازی "additional_information" تنها می‌تواند به عنوان یک آرایه با استفاده از روش معمول "Set[PROPERTY_NAME]" مقدار بگیرد. بنابراین نه تنها نمی‌توان آن را ایجاد کرد بلکه حتی قادر به مقداردهی آن به عنوان یک رشته نیز نخواهیم بود.

اما به روش زیرکانه‌ی دیگری می‌توان آن را مقداردهی کرد. وقتی مجنتو ویژگی‌های نمونه‌ی آرگمان را مقداردهی می‌کند، درواقع ویژگی‌های آن را مقداردهی نمی‌کند. درعوض آنها را در یک دیکشنری به نام "_data" ذخیره می‌کند. سپس این دیکشنری تقریبا در تمام مواردی که یک ویژگی نمونه مورد نیاز است مورد استفاده قرار می‌گیرد. در مثال گفته شده این بدان معنی‌ست که فیلد قابل سریال سازی "additional_information" درواقع داخل آن دیکشنری ذخیره می‌شود نه به عنوان یک ویژگی معمول. نتیجتا اگر بتوان دیکشنری "_data" را به طور کامل کنترل کرد، می‌توان از محدودیت آرایه فیلد "additional_information" جلوگیری کرد چون آن به طور دستی مقداردهی می‌شود نه با فراخوانی "Set[PROPERTY_NAME]". اما چگونه می‌توان این دیکشنری حساس را کنترل کرد؟

یکی از کارهایی که مجنتو قبل از ذخیره‌ی یک نمونه از "Payment" انجام می‌دهد ویرایش ویژگی‌های آن است. مجنتو این کار را با در نظر گرفتن ورودی API به عنوان اطلاعات پرداخت که لازم است در نمونه‌ی "Payment" ذخیره شود، انجام می‌دهد. این موضوع را می‌توان در متد API در شکل(4) مشاهده کرد.

[12] Property

```php
/**
 * Adds a specified payment method to a specified shopping cart.
 */
public function set($cartId, \Magento\Quote\Api\Data\PaymentInterface $method)
{

    $quote = $this->quoteRepository->get($cartId); // Get the cart instance
    $payment = $quote->getPayment(); // Get the payment instance

    // Get the data from the user input
    $data = $method->getData();

    // Check for additional data
    if (isset($data['additional_data'])) {
        $data = array_merge($data, (array)$data['additional_data']);
        unset($data['additional_data']);
    }

    // Import the user input to the Payment instance
    $payment->importData($data);

    ...
}

// PaymentMethodManagement::set()
```

شکل(4): تابع set()

هنگامی که مجنتو "getData()" را روی آرگومان "$method" فراخوانی می‌کند ویژگی "_data" از آن آرگومان برگردانده می‌شود که حاوی تمام اطلاعات پرداختی است که به عنوان ورودی آمده است. بعدا "importData()" را با ویژگی "_data" به عنوان ورودی فراخوانی می‌کند که این کار ویژگی "_data" از نمونه‌ی "Payment" را با ویژگی "_data" ی ورودی جایگزین می‌کند. اکنون مهاجم می‌تواند ویژگی حساس "_data" از نمونه‌ی "Payment" را با ویژگی "_data" که توسط مهاجم کنترل شده است جایگزین کند که این بدان معنی‌ست که می‌توان فیلد "additional_information" را مقداردهی کرد.

مشکل اینجاست که برای اینکه تابع "unserialize ()" بتواند کارکند، آن فیلد باید یک مقدار string بگیرد اما متد "Set[PROPERT_NAME]" فقط اجازه می‌دهد که آن فیلد یک آرایه باشد. راه حل این مشکل، دو خط کد قبل از فراخوانی "importData() " است. فیلد "additional_data" یک دیکشنری حاوی اطلاعات بیشتر برای روش پرداخت است که به طور کامل توسط کاربر کنترل می‌شود. به منظور اینکه داده‌های سفارشی بخشی از داده‌های اصلی باشند، مجنتو دیکشنری "additional_data" را با دیکشنری اصلی "_data" ادغام می‌کند. بدین ترتیب به دیکشنری "additional_data" اجازه داده می‌شود تا مقادر دیکشنری اصلی "_data" را نادیده بگیرد. این بدان معنی‌ست که بعد از ادغام دو دیکشنری، حالا دیکشنری "additional_data" می‌شود دیکشنری آرگمان "_data" و به دلیل فراخوانی "importData() "، ویژگی حساس "_data" از نمونه Payment برابر با دیکشنری "additional_data" خواهد شد. بدین صورت مهاجم می‌تواند فیلد قابل سریال سازی "additional_information" را به طور کامل کنترل کرده و از آن برای یک حمله تزریق شی استفاده کند.

4-3-1 قابلیت بهره برداری

بهره برداری از این آسیب‌پذیری می‌تواند از راه دور انجام گیرد. مهاجم می‌تواند با کنترل یک فیلد قابل سریال سازی و دسترسی به یک دایرکتوری قابل نوشتن توسط سرور قربانی، یک حمله‌ی تزریق شی را انجام دهد.

4-3-2 سطح آسیب پذیری

CVSS ریسک آسیب‌پذیری‌های مختلف را از 0.1 تا 10 به عنوان درجه low تا high درجه‌بندی می‌کند. این سایت به این آسیب‌پذیری امتیاز 7.5 را اختصاص داده و درجه آن را High ارزیابی کرده است.

5 مکانیزم کارکرد

در ادامه روش استفاده از آسیب‌پذیری مذکور شرح داده می‌شود.

5-1 خلاصه از حمله با استفاده از آسیب‌پذیری اجرای کد از راه دور

طبق توضیحات بخش قبل، حالا مهاجم می‌تواند هر رشته‌ای را که می‌خواهد unserialize کند.

ابتدا به یک شی با متد " __destruct()" نیاز است، بنابراین وقتی آن شی unserialize می‌شود و یا از بین می‌رود این متد به طور خودکار فراخوانی می‌شود. توجه کنید که به این متدها حتما نیاز است چون اگرچه می‌توان ویژگی‌های یک شی را کنترل کرد اما نمی‌توان هیچ کدام از متدهای آن شی را فراخوانی کرد. به همین دلیل باید به متدهای PHP که به طور خودکار زمانی رخ می‌دهند که وقایع خاصی اتفاق می‌افتد، تکیه کرد.

اولین شی‌ایی که استفاده می‌شود نمونه‌ای از کلاس "Credis_Client" است که متدهای آن را در شکل(5) می‌بینید.

```php
/*
 * Called automaticlly when the object is destrotyed.
 */
public function __destruct()
{
    if ($this->closeOnDestruct) {
        $this->close();
    }
}

/*
 * Closes the redis stream.
 */
public function close()
{
    if ($this->connected && ! $this->persistent) {
            ...
            $result = $this->redis->close();
    }
    ...
}

// Credis_Client::__destruct(), close()
```

شکل(5): متدهای کلاس "Credis_Client"

همانطور که مشاهده می‌شود این کلاس یک متد ساده "__destruct()" دارد (که زمانی که آن شی از بین می‌رود به طور خودکار توسط PHP فراخوانی می‌شود) که به سادگی متد "close()" را فراخوانی می‌کند. اگر "close()" تشخیص دهد که یک اتصال فعال به سرور ردیس[۱۳] وجود دارد، سعی می‌کند با فراخوانی متد "close()" در ویژگی "redis" آن اتصال فعال را ببندد.

از آنجا که تابع "unserialize()" اجازه‌ی کنترل همه‌ی ویژگی‌های یک شی را می‌دهد، ویژگی "redis" را نیز می‌توان کنترل کرد. به همین دلیل می توان هر شی دلخواهی را درون آن ویژگی (نه فقط "redis") قرار داد و هر متد "close()" را در هر کلاسی در آن سیستم فراخوانی کرد. در مجنتو چندین متد "close()" وجود دارد و چون متدهای "close()" معمولا مسئول پایان دادن به جریان‌ها، بستن فایل و ذخیره داده های شیءهستند، می‌توان فراخوانی‌های جالبی در آن پیدا کرد. در شکل(6) متد "close()" از کلاس "Transaction" نشان داده شده است.

[13] Redis server

```php
/**
 * Close this transaction
 */
public function close($shouldSave = true)
{
    ...
    if ($shouldSave) {
        $this->save();
    }
    ...
}

/**
 * Save object data
 */
public function save()
{
    $this->_getResource()->save($this);
    return $this;
}

// Magento\Sales\Model\Order\Payment\Transaction::__destruct(), close()
```

شکل(6): متد "close()" از کلاس "Transaction"

هر دو متد نشان داده شده بسیار ساده هستند. متد "close()" متد "save()" را فراخوانی می‌کند که آن هم متد "save()" از ویژگی "_resource" را فراخوانی می‌کند. با استفاده از منطق مشابهی که قبلا ذکر شده، چون می‌توان ویژگی "_resource" را کنترل کرد، می‌توان کلاس آن را نیز کنترل کرد و بنابراین می‌توان متد "save()" را در هر کلاس دلخواه کنترل کرد.

متدهای "save()" معمولا مسئول ذخیره برخی از انواع داده ها در برخی از انواع حافظه‌های ذخیره سازی (مانند سیستم فایل، پایگاه داده و غیره) هستند. شکل(7) یک متد "save()" که از سیستم فایل به عنوان حافظه‌ی ذخیره سازی استفاده می‌کند را نشان می‌دهد.

```php
/**
 * Try to save configuration cache to file
 */
public function save()
{
    ...
    // save stats
    file_put_contents($this->getStatFileName(), $this->getComponents());
    ...
}

// Magento\Framework\Simplexml\Config\Cache\File::save()
```

شکل(7): بخشی از متد "save()"

این متد چیزهای درون فیلد "components" را درون یک فایل ذخیره می‌کند. چون مسیر آن فایل از فیلد "stat_file_name" برگردانده می‌شود و چون هردوی این فیلدها را می‌توان کنترل کرد، بنابراین می‌توان مسیر این فایل و محتویات این فایل را کنترل کرد که در نتیجه این امر منجر به آسیب پذیری نوشتن در فایل دلخواه می شود.

یکی از مسیرهایی که در همه‌ی نسخه‌های نصب شده‌ی مجنتو در دسترس است مسیر "/pub" است. مجنتو از این دایرکتوری برای ذخیره هر تصویر یا فایل های عمومی که ادمین آپلود می‌کند استفاده می‌کند و

بنابراین این دایرکتوری قابل نوشتن توسط آن سرور است. از همه مهمتر اینکه چون این دایرکتوری معمولا تصاویری که باید توسط مرورگر بازیابی شود را ذخیره می‌کند، از طریق یک رابط وب[14] عادی قابل دسترسی است. بنابراین مهاجم می‌تواند تنها با نوشتن چند خط کد PHP آن کد را بدون نیاز به احراز هویت[15] روی سرور اجرا کند.

6 اقدامات جهت کاهش شدت آسیب پذیری

تا به حال راهکاری برای کاهش این آسیب‌پذیری اعلام نشده است. اما این آسیب‌پذیری به طور رسمی در جدیدترین نسخه‌ها وصله شده است و بنابراین نسخه‌های بروز رسانی شده‌ی این نرم‌افزار فاقد این آسیب‌پذیری می‌باشند.

7 جمع بندی و نتیجه‌گیری

در این گزارش به بررسی آسیب‌پذیری اجرای کد از راه دور که منجر به دستیابی به اطلاعات کاربران توسط مهاجم و بدون احراز هویت می‌شود پرداختیم. با استفاده از این آسیب‌پذیری، مهاجم می‌تواند با کنترل یک فیلد قابل سریال سازی و دسترسی به یک دایرکتوری قابل نوشتن توسط سرور قربانی و مقداردهی خطرناک به ویژگی "_data" از نمونه‌ی "payment"، یک حمله‌ی تزریق شی را انجام دهد.

8 منابع

[1] http://www.securityfocus.com/bid/90724/info

[14] Web interface
[15] Unauthenticated

[2] http://securityaffairs.co/wordpress/47439/hacking/cve-2016-4010-magento-flaw.html

[3] http://netanelrub.in/2016/05/17/magento-unauthenticated-remote-code-execution/