УДК 621.3.078.4
О.В.Дудік
Луцький національний технічний університет

АСПЕКТИ БЕЗПЕКИ ПРОГРАМУВАННЯ В PHP

О.В.Дудік. Аспекти безпеки програмування на PHP. *В статті розглядається ризики при програмуванні на мові PHP на рівнях даних, системи, виконання програм .*
Ключові слова: *запит, система, мережа, дані, права доступу.*
Літ. –2.

А.В.Дудик. Аспекты безопасности программирования на PHP. *В статье рассматривается риски при программировании на языке PHP на уровнях данных, системы, выполнения программ.*
Ключевые слова: *запрос, система, сеть, данные, права доступа.*
Лит. – 2.

O.V.Dudick. Safety aspects of programming in PHP. *This article discusses the risks of programming in PHP on levels of data, systems, program execution.*
Keywords: *request, system, network, data, permissions.*
References – 2.

Актуальність досліджуваної теми полягає в дослідженні можливостей збою роботи веб-систем і ресурсів для ефективного запобігання несанкціонованого втручання в їх роботу.

Проблема полягає в тому, що неефективна побудова веб-ресурсу призводить до його вразливості з боку хакерів, а також невірної роботи при виникненні нестандартних ситуацій.

Для вирішення даної проблеми розглядаються основні аспекти ефективного безпечного програмування на PHP, а також аналізуються різні фактори впливу на дані, систему і т.д.

Ризики на рівні даних

Ризики зберігання даних - це ті ризики, які включають розміщення даних у базі даних або файловій системі. Найбільш широко відомі атаки на цей клас - атаки **SQL injection**. Цей тип атак являє собою атаку в SQL-запит довільних даних, що може дозволити отримати практично будь-яку інформацію з бази даних аж до авторизаційної інформації і кредитних карт.

Найпростіший спосіб уникнути цих атак - це захистити кожну змінну, що використовується, яка використовується для SQL-запиту. На щастя, PHP має декілька вбудованих функцій для цього, наприклад **mysql_escape_string()**. Функція екранує усі спецсимволи в **unescaped_string**, внаслідок чого її можна безпечно вико-ристовувати в SQL-запиті.

Приклад екранування змінної:

```
$item = "Zak' s Laptop";
$escaped_item = mysql_escape_string($item);
printf ("Escaped string: %s\n", $escaped_item);
```

Вищеописаний приклад видасть наступний результат:

```
Escaped string: Zak\'s Laptop
```

Деякі програмісти вважають за краще екранувати дані, як тільки вони поступають в прикладенння, інші віддають перевагу робити це безпосередньо перед запитом у базу даних. Хорошим стилем вважається ставити екранування перед запитом, тобто можна завжди подивитися код, операції з базою даних і дані, які переходять в екранований запит, і не треба шукати за усім початковим кодом місце екранування.

Другий ризик, - це розміщення паролів в текстовому файлі (маються на увазі в незашифрованому вигляді). Багато прикладнень є з відкритим початковим кодом, де паролі знаходяться у відкритому вигляді. Необхідно узяти на замітку, що немає жодної вагомої причини залишати паролі у відкритому вигляді. Не має значення, де зберігаються паролі - в текстовому файлі або у базі даних, завжди необхідно їх розміщувати у вигляді хеша.

Трансформацію паролів досить просто зробити за допомогою PHP - функція **md5()**, слід зашифрувати їх перед збереженням. Оскільки md5 повторювана функція, можна перевіряти паролі звичайним порівнянням. MD5-хеш рядка str обчислюється, використовуючи алгоритм **MD5 RSA Data Security, Inc**. Хеш є 32-значним шістнадцятковим числом. Якщо необов'язковий аргумент **raw_output** має значення TRUE, то повертається бінарний рядок з 16 символів. (Зауваження: необов'язковий аргумент **raw_output** був доданий в PHP 5.0.0 і за умовчанням рівний FALSE.)

Не можна дозволяти змінній з паролем "плавати" по прикладненню. Як тільки пароль був введений, він відразу має конвертуватись. Бажано відразу помістити хеш пароля в змінну і використовувати тільки її.

Слід завжди розділяти про імена користувачів і паролі в різні PHP-файли з кодом і посилатися на них як на константи або змінні. Це не лише спростить програмування, але і, приміром, якщо необхідно змінити пароль, буде відоме його точне місце знаходження.

Приклад 1. Приклад використання md5():

```
$str = 'apple';
if (md5($str) === '1f3870be274f6c49b3e31a0c6728957f'){
echo "Would you like a green or red apple"?;
exit;
}
```

Тут результат md5($str) є зашифрованим рядком apple.

Також хотілося б торкнутися концепції розподілу прав. Усі користувачі бази даних повинні мати найменші права доступу, необхідні тільки для коректного виконання функцій. Якщо прикладненню потрібне тільки читання з бази даних, воно повинне мати права на виконання SELECT-запиту і ніяких прав доступу до інших баз даних.

Для дотримування цієї концепції рекомендується зробити декілька акаунтів у базі даних. Один акаунт матиме тільки права на запис в потрібні таблиці - INSERT, і повністю відокремлений акаунт матиме права на читання - SELECT. Це дозволить бути упевненим, що ніякі INSERT-запити не будуть випадково виконані, і це зменшить можливі ушкодження, зроблені ін'єкціями SQL.

Звичайно, декілька акаунтів працюють краще, коли є чіткий розподіл між тими, хто може писати у базу даних, і тими, хто може її читати (це використовується в деяких **CMS - Content Management Systems**). У теорії можна використовувати декілька акаунтів у будь-якому прикладненні, але можуть виникнути проблеми з численними підключеннями до бази даних, які мають бути вирішені на стадії дизайну архітектури ПЗ.

Програмуючи великі прикладнення, зручно розбивати код на безліч логічних файлів, але більшість з PHP-програмістів має звичку давати файлам розширення, відмінні від **.php**, наприклад **.inc** або **.config**. Це дуже погана ідея, томущо сервер може бути не налаштований на парсинг цих файлів як PHP, і абсолютно будь-хто може завантажити ці файли як початковий код (що потенційно містить паролі, імена користувачів і іншу приватну інформацію). Як варіант для наочності можна називати такі файли inc_ або class_ при необхідності, але розширення все-таки ставити .php.

Говорячи про інклудінг (включення, підвантаження) файлів, хотілося б також зазначити особливості безпечної роботи. Якщо наявний PHP-файл необхідно використовувати тільки як частину великого прикладнення, то вставляють його рядки в початок файлу (**__FILE__, $_SERVER['PHP_SELF']**).

Це захистить файл від прямого запуску, іншими словами, зупинить його роботу, якщо хтось звернеться до нього без-посередньо. Добре написаний клас або інклуд-файл не повинен робити нічого самостійно, але особливо з цього приводу можна не турбуватися - один раз вставлений рядок на початку файлу знімає усі проблеми.

Інша особливість інклудинга файлів - різниця між функціями **include()** і **readfile()**. Include говорить серверу парсити файл як PHP, тоді як readfile() інтерпретує файл як звичайний текст.

Ніколи не слід використовувати include з файлами, у яких є публічний доступ на запис, - наприклад, коли у вас є прикладнення, яке додає прийняті від користувача дані в кінець файлу (такі як гостьова книга), або з файлами, які ви не контролюєте (файли на іншому сервері або з можливістю редагування іншими користувачами), — довільний користувач зможе з легкістю впровадити його власний PHP-код у вашу систему.

В той же час ніколи не слід виконувати readfile з файлами, які закінчуються на .php. При неправильній конфігурації системи виникає ризик розкриття усім вашого початкового коду. Підвівши підсумки, можна сказати, що виконувати readfile() можна з html -, текстовими і видаленими файлами. Include() варто виконувати з локальними php-файлами, які необхідно виконати.

Ризики на рівні системи

Системний ризик залежить від того, як саме виконується код. Основний ризик системи у будь-якому прикладненні - невірні дані. Важко абсолютно і достовірно перевірити дані. Як тільки користувач вводить дані в систему, необхідно відразу ці дані перевіряти і фільтрувати. При перевірці даних треба активувати найретельніший фільтр. Приміром, якщо програма має на увазі відсотки, не треба перевіряти, чи було введено що-небудь, а перевіряти чітку відповідність числам між 0 і 100.

Також варто робити перевірки на усіх рівня коду. Кожного разу як функція отримує дані, перевіряють дані і, якщо вони невірні чи підозрілі, реагують відповідно. Це також дозволить уникнути логічних помилок у розробленому прикладненні.

Ризики на рівні виконання

Небезпечні функциії **eval()**, **exec()** і їм подібні (**shell_exec()**, **system()**, **passthru()** і **pcntl - exec()**).

Eval виконує будь-який переданий в його змінну php-код. Це досить небезпечно, тому що вже не буде абсолютного контролю над виконуваним кодом. Якщо все-таки необхідно використовувати **eval()**, ніколи не дозволяють йому запустити змінні, в яких дані отримуються від користувача, інакше хакер зможе впровадити свій код. **Exec()** і подібні команди містять аналогічні загрози, що дозволяють скрипту взаємодіяти з командним рядком, і варто використовувати цю можливість якомога рідше.

Існують ризики розкриття інформації. Тому варто прибирати повідомлення про помилки і налагоджувальну інформацію, бо вони можуть дати хакерам інформацію про потенційні помилки в системі. У працюючих системах завжди вимикають повідомлення про помилки і використовують замість цього функцію **PHP errorlog()**.

Останній ризик - це використання сесійних ідентифікаторів. Простіше кажучи, не можна посилати id-сесії користувачеві. Сесії не є небезпечними, але якщо передається сесійний id, то хтось інший, відмінний від очікуваного користувача, так званий «man in the middle», може оволодіти сесією користувача. Прикладом може бути перехоплення сесії інтернет-магазину. Зловмисник може отримати дані кредитної карти, змінити адресу доставки або зробити ще дещо гірше залежне від системи.

У статті обговорено основні ризики безпеки при програмуванні в PHP, але усі вони вписуються в декілька простих концепцій:

- ніколи не можна довіряти користувачеві, не дозволяти йому запускати код на сервері і завжди перевіряти дані, що надходять;
- не можна давати користувачеві або програмному забезпеченню рівень доступу більший абсолютного мінімуму, необхідного для успішного виконання завдань;
- не можна розкривати користувачеві більше інформації, чим йому необхідно знати, не дозволяти бачити початковий код, сесійні ID і будь-які повідомлення про помилки, за винятком тих, які створені спеціально для нього.

Реалізація захисту засобами АОП

Термін аспектно-орієнтоване програмування уперше був згаданий в роботі Gregor Kiczales et al. "Aspect - oriented programming" в 1997 році. АОП ставить своєю метою розробити механізм реалізації наскрізної функціональності в ООП системах. АОП не в якій мірі не замінює собою ООП, ця техніка усього лише добудовує концепції ООП. Для реалізації прикладів статті використовувався AspectJ - реалізація АОП для Java. У загальних словах АОП надає новий механізм композиції відмінний від наявних в ООП. Для прикладу розглянемо терміни, в яких працює AspectJ :

- JointPoint - певна точка у виконанні програми, це може бути виконання методу, зміна атрибуту класу, виклик методу, викидання виключення і так далі.
- Pointcut - набір (0.N) точок виконання програми, наприклад виконання усіх методів що починаються з «get» класів певного пакету, або виклик методів класів тих, що реалізовують деякий інтерфейс.
- Advice - java код виконуваний до (before advice), потім (after advice) або замість (around advice) кожної точки виконання що входять в певний pointcut.
- Aspect - модуль в термінах АОП, аналог класу в Java може містити публічні/приватні pointcut і advice крім того звичайні методи класу, можуть наслідувати і бути абстрактними.

• Introduction - метод зміни структури спадкоємстві і реалізацій існуючої системи, застосовується наприклад для того, щоб додати додатковий інтерфейс до існуючого класу або змінити ланцюг спадкоємців.

Загалом, аспекти додають додаткову функціональність в точки виконання програми (pointcut) через advice. Важливе те, що pointcut можуть збирати не лише точки виконання, але і контекст в яких ці точки знаходяться, наприклад, якщо pointcut визначає усі виклики методів someMethod усіх об'єктів класу SomeClass, то контекст виконання це:

- Об'єкт, метод якого викликаний.
- Параметри методу.
- Об'єкт, що викликав метод.

Приклад реалізації

```
package aop.example;

import aop.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.IOException;

/**
 * Авторизаційний аспект
 * @author Zubairov Renat
 */
public aspect AuthorizationAspect {

		/**
		 * pointcut метод, що включає
 з ServletRequest для
		 * того що б потім можна було
 би отримати його в об'єднанні
		 * з іншим pointcut (патерн "червоточина")
		 */
		pointcut requestMethod
(ServletRequest request) :
				execution(* aop.example.EntranceFilter.
doFilter(ServletRequest, ServletResponse,
FilterChain))
				&& args(request, ServletResponse
 FilterChain);

		/**
		 * pointcut визначальний метод фільтру
 у якому ми відловлюватимемо
		 * виключення
		 */
		public pointcut doFilterMethod
(ServletRequest srequest, ServletResponse
 sresponse, EntranceFilter filter) :
				execution(void aop.example.
EntranceFilter.doFilter(ServletRequest
 ServletResponse, FilterChain))
						&& args(srequest
 sresponse, FilterChain) && this(filter);


		// усі методи проводять читання інформації
 об'єктів моделі
		pointcut readMethods(Object object)
 : execution (public * aop.example.model.
*.get*(.)) && this(object);

		// усі методи проводять додавання
об'єктів моделі
		pointcut addMethods(Object object)
 : execution (public * aop.example.model.
*.add*(.)) && this(object);

		// усі методи проводять видалення
 об'єктів моделі
		pointcut deleteMethods(Album album)
 : execution (public * aop.example.
model.AlbumList.deleteAlbum(Album)) && args(album);

		// методи перевірки на доступність
читання (пред-перевірка)
```

```
		pointcut controlledRead(Object object)
 : execution(public boolean aop.example.model.
Controlled+.isReadable()) && this(object);

		// методи перевірки на доступність
додавання (пред-перевірка)
		pointcut controlledAdd(Object object)
 : execution(public boolean aop.example.model.Controlled+.
isAddable()) && this(object);

		// методи перевірки на доступність
видалення (пред-перевірка)
		pointcut controlledDelete(Object object)
 : execution(public boolean aop.example.model.
Controlled+.isDeletable()) && this(object);

		// Виклик методів читання що сталися в потоці
 виконання
		// що йде за викликом методу фільтру
		// ми об'єднали два pointcut - реалізація
патерну "червоточина"
		pointcut readAccess(ServletRequest request
 Object object) :
			cflow(requestMethod(request))
&& readMethods(object);

		// те ж саме тільки для додавання
		pointcut addAccess(ServletRequest request
 Object object) :
			cflow(requestMethod(request)) &&
addMethods(object);

		// те ж саме тільки для видалення
		pointcut deleteAccess(ServletRequest request
 Album album) :
			cflow(requestMethod(request))&&
 deleteMethods(album);

		// пред-перевірка на читання
		pointcut readCheck(ServletRequest request
 Object object) :
			cflow(requestMethod(request))&&
 controlledRead(object);

		// пред-перевірка на додавання
		pointcut addCheck(ServletRequest request
 Object object) :
			cflow(requestMethod(request))&&
 controlledAdd(object);

		// пред-перевірка на видалення
		pointcut deleteCheck(ServletRequest request
 Object object) :
			cflow(requestMethod(request))&&
 controlledDelete(object);

		/**
		 * Around advice що відловлює виключення
		 * і відправляючий запит на сторінку з помилкою
		 */
		void around(ServletRequest srequest
 ServletResponse sresponse, EntranceFilter filter)
 throws IOException, ServletException:
 doFilterMethod(srequest, sresponse, filter){
			try {
				// виконуємо метод фільтру
				proceed(srequest, sresponse, filter);
			} catch (AuthorizationException e){
				// ловимо виключення
				srequest.setAttribute("error_message",
e.getMessage());
				// вперед на сторінку з повідомленням
 про помилку
				filter.getConfig().getServletContext()
.getRequestDispatcher("error.vm").
forward(srequest, sresponse);
			}
		}
```

```
        /**
         * Before advice перевірки на читання
         */
        before(ServletRequest request, Object object)
 : readAccess(request, object){
                if (!AuthHelper.isAbleToRead
(extractUser(request), object)) {
                        throw new AuthorizationException
("Read access not allowed");
                }
        }

        /**
         * Before advice перевірки на додавання
         */
        before(ServletRequest request, Object object)
 : addAccess(request, object){
                if (!AuthHelper.isAbleToAdd
(extractUser(request), object)) {
                        throw new AuthorizationException
("Add access not allowed");
                }
        }

        /**
         * Before advice перевірки на видалення
         */
        before(ServletRequest request, Album album)
 : deleteAccess(request, album){
                if (!AuthHelper.isAbleToDelete
(extractUser(request), album)) {
                        throw new AuthorizationException
("Delete access not allowed");
                }
        }

        /**
         * Around advice пред-перевірки, тут
ми ігноруємо повернене методом
         * значення, і увесь час повертаємо
те яке задовольняє правилам
         * авторизації
         * Ми не обробляємо інші пред-перевірки
 оскільки за умовчанням будь-хто може
         * читати, і усе аутентифіковані
користувачі можуть додавати
         */
        boolean around(ServletRequest request
 Object object) : deleteCheck(request, object){
                return AuthHelper.isAbleToDelete
(extractUser(request), object);
        }

        /**
         * Приватна фунція, яка виймає
 користувача із запиту
         */
        private AnonymousUser extractUser
(ServletRequest request) {
                return (AnonymousUser)((HttpServletRequest)
 request).getSession().getAttribute
(EntranceFilter.USER_KEY);
        }

}
```

Висновки

Для ефективної роботи веб-систем необхідно :

1. Оцінити ризики на рівнях даних, систем і виконання.
2. Використовувати прогресивну аспектно-орієнтовану модель програмування.

1. Д.Н. Колисниченко. PHP 5 Самовчитель - Видання 3-і - Спб.: Наука і Техніка, 2006. - 576 с.
2. Ед Леки-Томпсон. PHP 5 для професіоналів. - М.: Вільямс, 2006.- 608с.