Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Extrakcia obsahu premenných z obrazu procesu v pamäti pre interpretovaný programovací jazyk PHP

Bakalárska práca

2017
Martin Sýkora

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Extrakcia obsahu premenných z obrazu procesu v pamäti pre interpretovaný programovací jazyk PHP

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Richard Ostertág, PhD.
Konzultant:	Mgr. Peter Košinár

Bratislava, 2017
Martin Sýkora

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Martin Sýkora
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Extrakcia obsahu premenných z obrazu procesu v pamäti pre interpretovaný programovací jazyk PHP
Variables content extraction from process core-dump for interpreted programming language PHP

Cieľ: Cieľom práce je naštudovať spôsob ukladania obsahu premenných pre vybraný interpretovaný programovací jazyk PHP (verzia 5 a 7) a využiť tieto vedomosti pre implementáciu nástroja, ktorý bude schopný z obrazu procesu vypísať hodnoty všetkých premenných existujúcich v danom okamihu v interpretovanom programe.

Vedúci: RNDr. Richard Ostertág, PhD.
Konzultant: Mgr. Peter Košinár
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 25.10.2016

Dátum schválenia: 25.10.2016 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Ďakujem môjmu školiteľovi RNDr. Richardovi Ostertágovi, PhD. a konzultantovi Mgr. Petrovi Košinárovi za pomoc a rady pri práci. Ďakujem mojej rodine a priateľom za podporu pri vypracovávaní tejto práce.

Abstrakt

Táto bakalárska práca sa zaoberá spôsobom ukladania premenných a ich hodnôt v interpretovanom programovacom jazyku PHP vo verziách 5 a 7. Keďže výsledná aplikácia pracuje s obrazom procesu v pamäti, ktorý je v našom prípade uložený v súborovom formáte ELF, tak práca obsahuje aj popis tohto formátu.

Kľúčové slová: premenná, extrakcia, obraz procesu v pamäti, PHP 5, PHP 7

Abstract

This bachelor thesis deals with method of storing variables and their content in interpreted programming language PHP in versions 5 and 7. Created application uses core dump, which is in our case saved in ELF file, therefore this paper contains description of this format.

Keywords: variable, extraction, core dump, PHP 5, PHP 7

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam ukážok kódu

Úvod

Žijeme v dobe, kedy počítače a počítačové systémy zasahujú stále viac a viac do všetkých oblastí každodenného života. Prostredníctvom počítačových systémov dochádza k obrovskému množstvu počtu pohybu finančných prostriedkov, uchovávaniu, archivácii a výmene citlivých informácií (osobných, firemných a pod.). Okrem komunikácie medzi ľuďmi dochádza aj ku komunikácii medzi samotnými počítačovými systémami.

Z tohto hľadiska sú počítačové systémy zaujímavé pre ľudí, ktorí ich chcú zneužiť -- väčšinou na vlastné finančné a materiálne obohatenie sa, poškodenie druhej strany alebo len na dokázanie vlastných schopností tým, že preniknú cez zabezpečenie počítačových systémov. Z týchto dôvodov sa počítačové systémy často stávajú terčom útočníkov.

V súčasnosti sa s rastúcim výskytom škodlivého softvéru zvyšuje aj počet škodlivých programov napísaných v niektorom z interpretovaných jazykov.

Štandardným postupom pri výskyte neznámeho a potenciálne škodlivého procesu v systéme by bolo upravenie interpretera tak, aby logoval volania funkcií s ich parametrami a návratovou hodnotou do súborov. Tento prístup má dve hlavné nevýhody. Prvou z nich je skutočnosť, že takéto súbory môžu narásť do obrovskej veľkosti. Druhou, podstatnejšou skutočnosťou je, že pri tomto postupe je potrebné procesy využívajúce interpreter ukončiť, následne ho upraviť a znovu ho spustiť. Teda po upravení interpretera už nemusí inkriminovaný proces znovu vzniknúť.

Naša aplikácia bude brať ako vstup obraz pamäti daného procesu, ktorý je možné získať aj bez toho, aby proces skončil. Následne bude aplikácia tento obraz spracovávať.

Práca je členená na niekoľko častí. V kapitole 1 sa budeme venovať forenznej analýze pamäte a ukážeme niektoré nástroje, ktoré sa pri analýze pamäte využívajú. Rovnako uvedieme aj nástroje, pomocou ktorých možno získať obraz pamäte procesu.

V kapitole 2 si zadefinujeme niekoľko pojmov súvisiacich so spôsobom vykonávania niektorých typov programovacích jazykov. Uvedieme aj niektoré kritéria, podľa ktorých sú jazyky rozdelené, vrátane príkladov takýchto jazykov.

Keďže naša aplikácia bude používať ako vstup súbor typu ELF, v kapitole 3 si popíšeme štruktúru tohto súborového formátu.

V kapitole 4 popíšeme spôsob fungovania referenčnej implementácie PHP, uvedieme aj alternatívy referenčnej implementácie. Následne popíšeme spôsob ukladania premen-

ných v referenčnej implementácii PHP vo verziách 5 a 7.

Kapitola 5 obsahuje implementačné detaily aplikácie, rovnako ako aj návrhy na jej možné rozšírenia a vylepšenia.

Súčasťou tejto bakalárskej práce je aj elektronický nosič CD, na ktorom je uložený zdrojový kód výslednej aplikácie.

Kapitola 1

Forenzná analýza pamäte

V tejto kapitole vysvetlíme pojem forenzná analýza pamäte a predstavíme nástroj, ktorý sa pri jej vykonávaní používa najčastejšie. Uvedieme spôsoby, akými možno získať obraz procesu v pamäti pre operačné systémy Windows a Linux.

Závislosť sveta na počítačových systémoch rastie každý deň. Spoločnosti sa chránia pomocou digitálnej obrany ako firewall, šifrovanie a skenovanie na základe signatúr. Navyše sú plánované útoky aj na elektrické rozvodné siete, infiltrovanie vojenských dátových centier a kradnutie obchodných tajomstiev verejných a súkromných spoločností.

1.1 Forenzná analýza pamäte

Ak dôkaz kompromitácie nie je zapísaný na disk, tak nám forenzná analýza disku nepomôže. Na druhej strane je vysoká pravdepodobnosť, že operačná pamäť bude obsahovať celý alebo časť škodlivého kódu, hoci nikdy nebude zapísaný na disk, pretože musí byť načítaný v pamäti, aby mohol byť vykonaný.

Každá akcia vykonaná operačným systémom alebo nejakým programom vyústi v špecifickú zmenu počítačovej pamäte, ktorá často zostáva dlho nezmenená po vykonaní akcie. Navyše forenzná analýza pamäte poskytuje bezprecedentný pohľad do stavu systému počas behu ako bežiace procesy, otvorené spojenia so sieťou a nedávno vykonané príkazy. Významné dáta často existujú výhradne iba v pamäti, ako napríklad šifrovacie kľúče disku, injektované kusy kódu, dešifrované e-maily a neuchovávané záznamy histórie prehliadania internetu.

Forenzná analýza pamäte je teda v podstate vyhodnotenie a interpretácia rozsiahleho objemu dát s použitím techník vizualizácie a extrakcie. Viac o forenznej analýze pamäte sa možno dočítať v knihe *The Art of Memory Forensics* [9].

1.2 Nástroje pre zber

V tejto práci sa budeme venovať menším častiam pamäte, konkrétne obrazom procesov, v ktorých bude spustený interpreter jazyka PHP vo verzii 5 alebo 7.

Na získanie obrazu procesu v pamäti možno použiť pre operačný systém Windows nástroj ProcDump od spoločnosti SysInternals a pre operačný systém Linux nástroj gcore.

1.2.1 ProcDump

ProcDump [15] je nástroj na vytváranie obrazu procesu z pamäte, ktorý pracuje podobne ako nižšie spomínaný program gcore. ProcDump dokáže vytvoriť dva druhy obrazu procesu z pamäte - FullDump a MiniDump. FullDump obsahuje celú vyextrahovanú pamäť procesu, zatiaľ čo MiniDump obsahuje len pamäť obsahujúcu zásobník hlavného vlákna (*threadu*) a jeho ovládačov (*handlerov*), neobsahuje teda napríklad globálne premenné.

Výstupy programu ProcDump možno analyzovať pomocou nástroja WinDbg, a tak získať obsahy premenných. Aby spomínaný nástroj fungoval, je nutné mať k dispozícii debugovacie symboly aplikácie, z ktorej bol obraz pamäte vytvorený.

1.2.2 Gcore

Gcore [14] (*generate a core file*) je program systému UNIX na vytváranie obrazu procesu v pamäti (*core dump*) špecifikovaním identifikátora procesu PID (*Process Identifier*). Výstupný súbor je ekvivalentný súboru, ktorý by vygenerovalo jadro (*kernel*), ak by proces neočakávane skončil. Po skončení programu gcore zostáva proces bežať bez zmeny. Formát súboru vygenerovaný programom gcore je štandardný linuxový formát ELF (*Executable and Linkable Format*), ktorý si predstavíme v časti 3.

1.3 Nástroje pre analýzu

1.3.1 Volatility framework

Momentálne najpoužívanejším nástrojom na forenznú analýzu pamäte (*volatile memory*) je framework Volatility s otvoreným zdrojovým kódom. Prvá verzia frameworku Volatility bola vydaná v roku 2007. Softvér bol založený na množstve akademických publikácií o pokročilej analýze pamäte. Do tej doby bola digitálna forenzná analýza zameraná na analýzu pevných diskov. Volatility ukázalo „silu“ analyzovania bežiaceho systému pomocou dát prítomných v operačnej pamäti. Od toho času sa stala analýza pamäte jednou z najdôležitejších tém digitálneho vyšetrovania a Volatility sa stalo

najpoužívanejšou platformou na svete na analýzu pamäte. Taktiež poskytuje medziplatformovú, modulárnu a rozšíriteľnú platformu pre výskum v tejto oblasti. Stalo sa nevyhnutným nástrojom digitálneho vyšetrovania a spoliehajú sa naň orgány činné v trestnom konaní, vojenskí, akademickí a komerční odborníci v oblasti analýzy pamäte na celom svete.

Techniky extrakcie sú vykonávané úplne nezávisle od platformy, na ktorej beží vyšetrovaný systém. Volatility nepodporuje vytváranie obrazov pamäte. Obrazy pamäte, ktoré je schopné spracovať, ako aj všetky pluginy, sú dostupné v zdrojovom repozitári projektu [5].

Kapitola 2

Stručná kategorizácia programovacích jazykov

V tejto kapitole si zadefinujeme niekoľko pojmov súvisiacich so spôsobom vykonávania niektorých programovacích jazykov. Uvedieme aj niektoré kritériá, podľa ktorých sú jazyky rozdelené a uvedieme k nim aj príklady.

2.1 Bajtkód

Bajtkód (*bytecode*) je binárna reprezentácia spúšťateľného programu určená k behu na virtuálnom stroji, nie na vyhradenom hardvéri. Keďže je bajtkód spracovávaný softvérovo, tak je zvyčajne viac abstraktný ako strojový kód. Na rozdiel od zdrojového kódu čitateľného pre ľudí, bajtkód je výsledok parsovania a sémantickej analýzy zdrojového kódu. Z uvedeného dôvodu je rýchlosť vykonávania programu vyššia ako priama interpretácia zdrojového kódu. Výhodou je aj prenositeľnosť. Medzi jazyky kompilujúce zdrojový kód do bajtkódu patria Java, C# a iné.

2.2 JIT kompilácia

JIT (*Just In Time*) kompilátor je súčasťou interpretera a za behu prekladá často interpretované časti kódu do strojového kódu počítača. Zvyčajne je to bajtkód alebo nejaký druh inštrukcií virtuálneho stroja. JIT kompilátor má prístup k informáciám počas behu, čím sa líši od štandardného kompilátora, a teda vie kód lepšie optimalizovať.

2.3 Rozdelenie programovacích jazykov podľa premenných

Podľa spôsobu, ako sú premenné deklarované, rozdeľujeme jazyky na dve základné skupiny:

- ***staticky typované*** jazyky musia mať v čase kompilácie určený typ premenných. Medzi takéto jazyky patria napríklad C++ a Java,
- ***dynamicky typované*** jazyky majú typ premenných odvodený počas behu programu. Medzi takéto jazyky patria napríklad PHP, Perl, Python a VBScript.

Existujú aj ďalšie kritériá, pre spôsob akým sú typy vynucované alebo ako sú medzi sebou konvertované. Poznáme:

- ***silno typované*** jazyky pri argumente nesprávneho typu generujú chybu. Všetky staticky typované jazyky sú silno typované, hoci mnohé povoľujú pretypovanie premennej. Medzi jazyky, ktoré sú dynamicky a silno typované, patria Python a Ruby.
- ***slabo typované*** jazyky nutne nevynucujú typy. Dosahujú to konverziou premenných na vhodný typ. Medzi takéto jazyky patria napríklad Perl, PHP a JavaScript.

Kapitola 3

Executable and Linkable Format

V tejto kapitole si popíšeme základnú štruktúru súboru vo formáte ELF a podrobnejšie si rozoberieme niektoré jeho časti, ktoré budeme potrebovať pri implementácii našej aplikácie.

Executable and Linkable Format (ELF) [17] je štandardný súborový formát pre spúšťateľné a objektové súbory, zdieľané knižnice a súbory s obrazom pamäte (*core file, core dump*). Názov ELF pôvodne znamenal *Extensible Linking Format*. ELF bol vytvorený a publikovaný organizáciou *Unix System Laboratories* (USL) ako časť *Application Binary Interface*. V roku 1999 bol zvolený za štandardný formát binárnych súborov pre UNIX-ové systémy.

3.1 Štruktúra

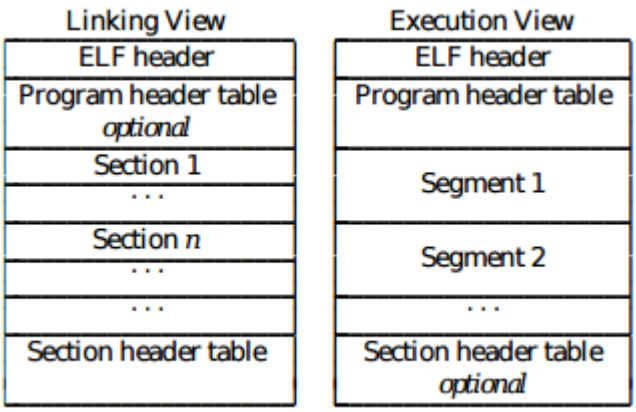
Súbor ELF vždy obsahuje ELF hlavičku (*ELF header*) a môže obsahovať tabuľku hlavičiek programu (*program header*) alebo tabuľku hlavičiek sekcií (*section header*), ktorá popisuje sekcie (napríklad .data, .text, .bss a iné).

Ako možno vidieť na obrázku 3.1, z pohľadu linkovania (*linking view*) je súbor rozdelený na sekcie. Tento pohľad je využívaný, keď program alebo knižnica je linkovaná. Sekcie obsahujú množstvo informácií o objektových súboroch pre pohľad linkovania, napr. dáta, inštrukcie, informácie o relokáciach, tabuľku symbolov, debugovacie informácie a iné.

Z pohľadu vykonávania (*execution view*) je súbor rozdelený na segmenty. Tento pohľad je použitý počas behu programu. Ak sú prítomné hlavičky programu, obsahujú informácie určené pre kernel (ako načítať segmenty do pamäte pomocou funkcie *mmap*).

3.1.1 ELF hlavička

ELF hlavička sa začína „magickými bajtmi“ 0x7f, 'E', 'L', 'F', ktoré identifikujú súbor ako ELF. Nasledujúci bajt udáva, či je program 32-bitový alebo 64-bitový. Ďalší



Obr. 3.1: Rozloženie súboru ELF z pohľadu linkovania (*linking view*) a z pohľadu vykonávania (*execution view*) [17]

bajt určuje, či bude súbor kódovaný vo formáte Little alebo Big endian. Medzi ďalšie významné položky ELF hlavičky patria `e_type`, `e_phoff`, `e_shoff`, `e_ehsize`, `e_phentsize`, `e_phnum`, `e_shentsize` a `e_shnum`.

Položka `e_type` určuje typ súboru, teda či ide o relokovateľný (*relocatable*), spustiteľný (*executable*), zdieľaný súbor (*shared*) alebo súbor (*core file, core dump*), ktorý je vygenerovaný pri neočakávanom skončení programu.

Offset v bajtoch od začiatku súboru k tabuľke programových hlavičiek udáva `e_phoff`. Ak súbor neobsahuje tabuľku programových hlavičiek, hodnota tejto položky je 0.

Položka `e_shoff` hovorí o offsete v bajtoch tabuľky hlavičiek sekcií od začiatku súboru. Ak súbor neobsahuje tabuľku hlavičiek sekcií, hodnota tohto člena je 0.

Veľkosť ELF hlavičky v bajtoch udáva položka `e_ehsize`.

Člen `e_phentsize` udáva veľkosť jedného záznamu v tabuľke hlavičiek programu v bajtoch. Každý záznam má rovnakú veľkosť. Člen `e_phnum` nám určuje počet záznamov v tabuľke hlavičiek programu. Veľkosť tabuľky hlavičiek programu je súčinom `e_phentsize` a `e_phnum`. Ak program neobsahuje tabuľku hlavičiek programu, hodnota tohoto člena je 0.

Analogicky, `e_shentsize` udáva veľkosť jedného záznamu v tabuľke hlavičiek sekcií v bajtoch, kde má každý záznam rovnakú veľkosť a `e_shnum` určuje počet záznamov v tabuľke hlavičiek sekcií. Veľkosť tabuľky hlavičiek sekcií je súčinom `e_shentsize` a `e_shnum`.

3.1.2 Hlavička programu

Tabuľka hlavičiek programu spustiteľného alebo zdieľaného objektového súboru je pole štruktúr, kde každá popisuje segment alebo inú informáciu, ktorú systém potrebuje, aby pripravil program pre vykonávanie. Segment objektového súboru obsahuje jednu

alebo viacero sekcií. Programové hlavičky majú zmysel iba pri spustiteľných alebo zdieľaných objektových súboroch. Hlavička programu zobrazená v ukážke kódu 3.1 má nasledujúcu štruktúru:

Ukážka kódu 3.1: Štruktúra programovej hlavičky

```
typedef struct {
        Elf64_Word      p_type;
        Elf64_Word      p_flags;
        Elf64_Off       p_offset;
        Elf64_Addr      p_vaddr;
        Elf64_Addr      p_paddr;
        Elf64_Xword     p_filesz;
        Elf64_Xword     p_memsz;
        Elf64_Xword     p_align;
} Elf64_Phdr;
```

- ***p_type*** určuje, aký druh segmentu táto štruktúra popisuje, alebo ako interpretovať informácie v štruktúre (hodnoty a ich významy popíšeme neskôr),
- ***p_flags*** určuje, či je segment čitateľný, vykonateľný alebo je doň možné zapisovať,
- ***p_offset*** určuje offset od začiatku súboru, na ktorom sa nachádza prvý bajt segmentu,
- ***p_vaddr*** určuje virtuálnu adresu prvého bajtu v segmente,
- ***p_paddr*** určuje na systémoch, kde je relevantné fyzické adresovanie, fyzickú adresu segmentu,
- ***p_filesz*** určuje veľkosť segmentu v obraze súboru (file image), môže byť nula,
- ***p_memsz*** určuje veľkosť segmentu v pamäti,
- ***p_align*** určuje hodnotu, na ktorú majú byť segmenty v súbore zarovnané. Nula a jedna znamenajú žiadne zarovnanie. Hodnotou `p_align` by malo byť kladné číslo, ktoré je mocninou dvojky a má platiť vzťah `p_vaddr` $\equiv$ `p_offset` (mod `p_align`)

Typy segmentov Typ segmentu je špecifikovaný v hlavičke programu v položke `p_type`. Všetky typy sú znázornené v tabuľke 3.1.
Typ `PT_LOAD` špecifikuje načítateľný segment (*loadable segment*), ktorý je popísaný

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOOS	0x60000000
PT_HIOS	0x6fffffff
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

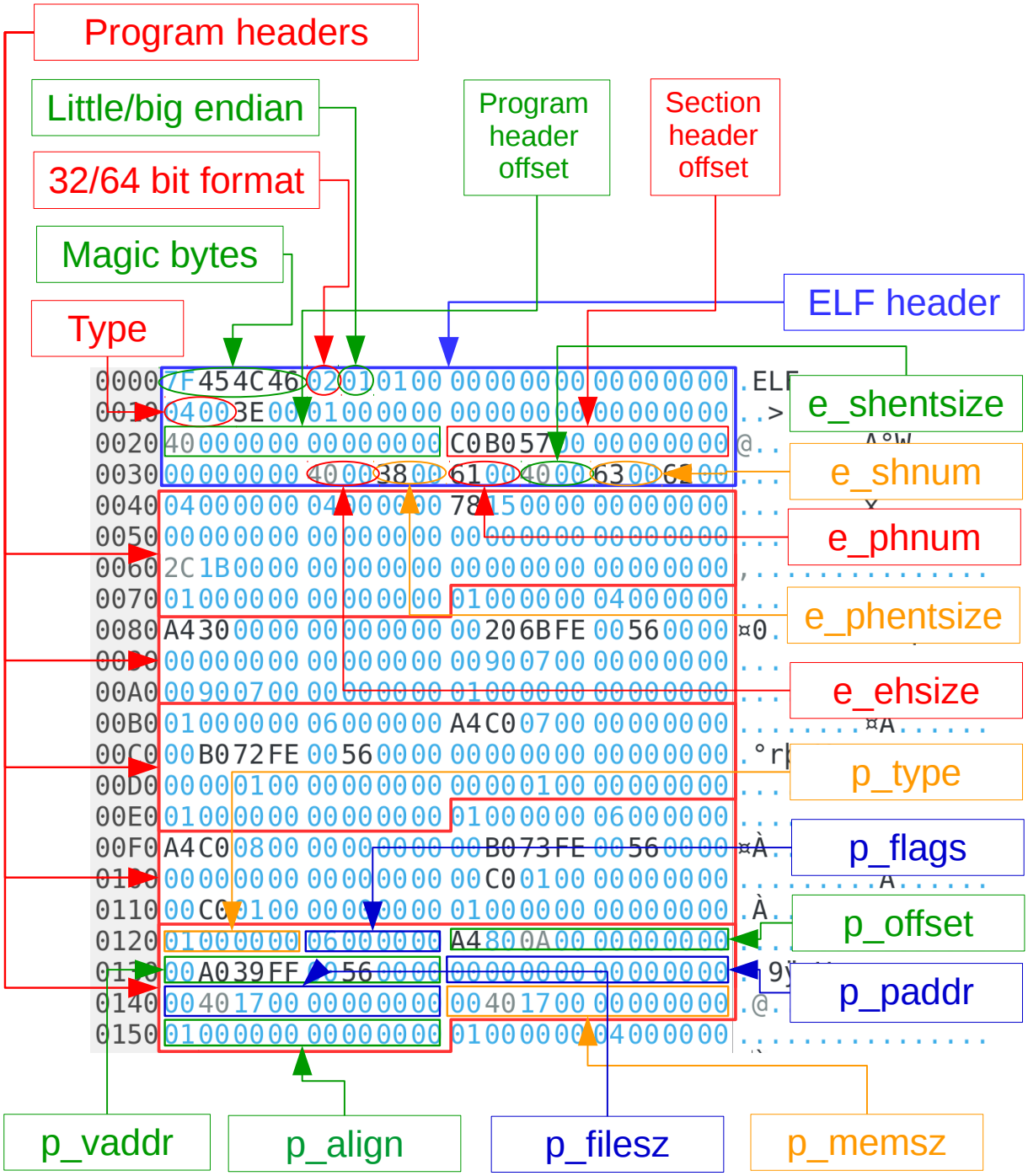
Tabuľka 3.1: Typy segmentov

```
Program Headers:
    Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
    NOTE           0x0011f8 0x0000000000000000 0x0000000000000000 0x001dfc 0x000000 R   0x1
    LOAD           0x002ff4 0x0000000000400000 0x0000000000000000 0x000000 0x7f3000 R E 0x1
    LOAD           0x002ff4 0x0000000000df3000 0x0000000000000000 0x0a1000 0x0a1000 R   0x1
    LOAD           0x0a3ff4 0x0000000000e94000 0x0000000000000000 0x00d000 0x00d000 RW  0x1
    LOAD           0x0b0ff4 0x0000000000ea1000 0x0000000000000000 0x01d000 0x01d000 RW  0x1
    LOAD           0x0cdff4 0x0000000001dbf000 0x0000000000000000 0x275000 0x275000 RW  0x1
    LOAD           0x342ff4 0x00007f54c52a5000 0x0000000000000000 0x000000 0x025000 R E 0x1
    LOAD           0x342ff4 0x00007f54c54c9000 0x0000000000000000 0x004000 0x004000 R   0x1
    LOAD           0x346ff4 0x00007f54c54cd000 0x0000000000000000 0x001000 0x001000 RW  0x1
    LOAD           0x347ff4 0x00007f54c54ce000 0x0000000000000000 0x000000 0x029000 R E 0x1
```

Obr. 3.2: Hlavičky programu, časť výstup programu readelf s prepínačmi -W a --segments

pomocou `p_filesz` a `p_memsz`. Bajty zo súboru sú namapované na začiatok pamäťového segmentu. Ak veľkosť pamäte segmentu (`p_memsz`) je väčšia ako veľkosť v súbore (`p_filesz`), tak „nadbytočné bajty“ budú nulové. Veľkosť v súbore by nemala byť väčšia ako veľkosť v pamäti. Záznamy spustiteľných segmentov sa v hlavičkách programu objavujú vo vzostupnom poradí, zoradené podľa `p_vaddr`.

Naša aplikácia bude ako vstup brať 64-bitový súbor ELF typu core vo formáte Big endian. Keďže obraz procesu vznikne počas behu programu, zaujímať nás budú segmenty, ktoré sú typu `PT_LOAD`.



Obr. 3.3: Súborový formát ELF

Kapitola 4

PHP

V tejto kapitole si predstavíme jazyk PHP, popíšeme spôsob ukladania premenných vo verziách 5 a 7 a uvedieme rozdiely medzi nimi.

PHP [7] je skriptovací jazyk s otvoreným zdrojovým kódom. Tento jazyk bol vytvorený Rasmusom Lerdorfom v roku 1995 a oficiálne predstavený v roku 1997. Používa sa najmä na programovanie klient-server aplikácií (na strane servera) a pre vývoj dynamických webových stránok. Dá sa použiť aj na tvorbu konzolových a GUI (*Graphical User Interface*) aplikácií. Názov PHP pôvodne znamenal Personal Home Page, ale dnes sa používa rekurzívny akronym PHP: Hypertext Preprocessor. Momentálne najpoužívanejšou verziou PHP je PHP 5 s podielom 94,3 % [18]. Avšak aktuálnou stabilnou verziou je od konca roka 2015 PHP 7, verzia 6 nikdy nebola stabilnou verziou. Verzia 7 priniesla zvýšenie rýchlosti aj za pomoci pozmeneného spôsobu ukladania premenných.

V tejto časti budeme čerpať informácie hlavne z knihy *Advanced PHP Programming* [16], online príručky PHP [12] a blogov, ktorých autormi sú Nikita Popov [13] a Julien Pauli [11].

Niektoré z ukážok kódu nie sú platné pre celú verziu, lebo časom boli zmenené niektoré členy štruktúr. Uvedené štruktúry sú platné pre PHP vo verziách 5.6.30 a 7.1.5, ktorá je v čase písania tejto práce aktuálnou verziou.

4.1 Zend Engine

Pôvodná, úplná a stále najrozšírenejšia implementácia PHP je postavená na Zend Engine [8], známa ako PHP. Na rozlíšenie od iných implementácií je niekedy označovaná ako Zend PHP. Zend Engine kompliluje PHP skript počas behu (on the fly) do internej reprezentácie, ktorú vie vykonať, teda pracuje ako interpreter.

Zend Engine je virtuálny stroj (virtual machine (VM)), čo znamená, že je to softvérový program, ktorý simuluje fyzický počítač. V jazyku ako Java, architektúra VM poskytuje portabilitu, čo dovoľuje prenášať skompilovaný bajtkód z jedného počítača

na druhý.

Na rozdiel od 75 základných inštrukcií procesorov architektúry x86, Zend Engine implementuje približne 150 základných inštrukcií. Táto inštrukčná sada nezahŕňa len inštrukcie typické pre VM, ako sú logické a aritmetické operácie, ale aj komplexné inštrukcie, ako je volanie procedúry `include` a výpis reťazca.

VM je vždy pomalší ako počítač, na ktorom beží. Nejaká rýchlosť je získaná vykonávaním komplexných inštrukcií ako jednej operácie VM.

4.1.1 Princíp fungovania

Zend Engine vykonáva skript tak, že prechádza nasledujúcimi krokmi:

1. Skript prejde cez lexikálny analyzér (nazývaný lexer), ktorý konvertuje kód z podoby čitateľnej pre ľudí do postupnosti tokenov, ktoré je schopný počítač spracovať.

2. Parser sparsuje postupnosť tokenov, ktoré získa od lexikálneho analyzéra a vygeneruje sadu inštrukcií, ktorá je určená na vykonávanie. Parser generuje abstraktný syntaktický strom, ktorý môže byť optimalizovaný pred tým, ako nastane generovanie kódu. Celý tento mechanizmus je nazývaný kompilácia. Výstupom kompilácie je bajtkód, ktorý je nezávislý na hardvéri.

3. Po vygenerovaní bajtkódu exekútor cezeň prechádza a vykonáva ho.

4.1.2 Úlohy

Zend Engine je v PHP zodpovedný za:

- parsovanie, zahŕňajúc kontrolu syntaxe,
- kompiláciu v pamäti (in memory compilation),
- vykonávanie bajtkódu,
- implementáciu všetkých štandardných dátových štruktúr PHP,
- vytváranie interfacu medzi modulmi pre konektivitu a protokolmi,
- preťaženie objektovo orientovanej syntaxe pre integráciu s Javou a .NET,
- poskytovanie všetkých štandardných služieb, zahŕňajúc manažment pamäte, manažment zdrojov ...

4.1.3 Alternatívy

Zend Engine je zároveň aj referenčná implementácia PHP. Keďže PHP nemá formálnu špecifikáciu, sémantika Zend PHP definuje aj sémantiku samotného PHP. Kvôli komplexnej sémantike, ktorá je definovaná tým, ako Zend PHP pracuje, je ťažké pre iné implementácie ponúknuť úplnú kompatibilitu.

Fakt, že Zend Engine je interpreter, spôsobuje celkovú neefektívnosť. V dôsledku tejto skutočnosti boli vyvinuté rôzne aplikácie na zlepšenie výkonu PHP. Aby sa zrýchlilo vykonávanie, môže byť skript predkompilovaný do formátu pre vykonávanie.

Zend PHP je stále najrozšírenejšiou implementáciou, hoci existujú aj iné implementácie. Niektoré z nich sú komplilátory alebo podporujú JIT kompiláciu. Medzi alternatívne implementácie patria napríklad:

- ***HipHop Virtual Machine (HHVM)*** [2] – vyvinutý spoločnosťou Facebook s verejne dostupným kódom. PHP skript transformuje do bajtkódu.
- ***Parrot*** [10] – virtuálny stroj (Virtual machine (VM)) s verejne dostupným kódom, určený na vykonávanie dynamických programovacích jazykov. PHP skript je pretransformovaný do Parrot Intermediate Language (PIR), ktorý je následne pretransformovaný do Parrot bajtkódu a následne je vykonaný.
- ***Phalanger*** [1] – kompiluje PHP skript do inštrukčnej sady frameworku .NET. Sú to logické jednotky obsahujúce CIL (Common Intermediate Language) bajtkód a metadáta.
- ***HipHop for PHP*** [3] – vyvinutý spoločnosťou Facebook s verejne dostupným kódom. Transformuje skript napísaný v jazyku PHP do kódu jazyka C++ a následne ho skompiluje do výsledného kódu. V roku 2013 bol jeho vývoj zastavený v prospech HHVM.

Hoci alternatívne implementácie ponúkajú vyšší výkon ako Zend PHP, nedokážu zabezpečiť úplnú kompatibilitu s PHP.

4.2 PHP 5

4.2.1 Zval

Keďže PHP je dynamicky typovaný jazyk implementovaný v staticky, silno typovanom jazyku C, je nutná konverzia medzi týmito dvomi jazykmi. Slúži na to štruktúra `zval_struct` (skratka zo *Zend value*), ktorá je znázornená v ukážke kódu 4.1. Položky `refcount__gc` a `is_ref__gc` slúžia na garbage collection. O počte referencií hovorí

Ukážka kódu 4.1: Zval v PHP 5

```
typedef struct _zval_struct zval;

struct _zval_struct {
        /* Variable information */
        zvalue_value value;             /* value */
        zend_uint refcount__gc;
        zend_uchar type;                /* active type */
        zend_uchar is_ref__gc;
};
```

Name	Value
IS_NULL	0
IS_LONG	1
IS_DOUBLE	2
IS_BOOL	3
IS_ARRAY	4
IS_OBJECT	5
IS_STRING	6
IS_RESOURCE	7
IS_CONSTANT	8
IS_CONSTANT_AST	9
IS_CALLABLE	10

Tabuľka 4.1: Dátové typy v PHP 5

`refcount__gc` a o tom, či `zval` je referencia `is_ref__gc`. Všetky typy premenných, ktoré PHP využíva sú vymenované v tabuľke 4.1. Samotná hodnota je uložená v člene `value`, ktorý je typu `zvalue_value` a je znázornený v ukážke kódu 4.2. Ide o variantný typ (*union*) `zvalue_value`, teda obsahom môže byť každý z uvedených typov. Tieto typy sa však prekrývajú, čo znamená, že vo variantnom type môže byť v jednom momente len jedna hodnota.

Ukážka kódu 4.2: Zvalue 5

```
typedef union _zvalue_value {
        long lval;                  /* long value */
        double dval;                /* double value */
        struct {
```

```
5                   char *val;
6                   int len;
7           } str;
8           HashTable *ht;          /* hash table value */
9           zend_object_value obj;
10          zend_ast *ast;
11  } zvalue_value;
```

Reťazce určuje štruktúra `str` a prislúcha im typ `IS_STRING`. Do dĺžky reťazca sa nezapočítava koncový nulový bajt z dôvodu, že tento bajt tam nie je prítomný, aby mohol byť použitý v rámci reťazca.

Premenné typu `IS_NULL` hodnotu neuchovávajú, pretože pre tento typ existuje iba jedna hodnota a tou je null. Desatinné čísla sú typu `IS_DOUBLE` a vo variantnom type `zvalue_value` sú uložené v položke `dval`. Typy `IS_LONG`, `IS_BOOL` a `IS_RESOURCE` sa ukladajú ako `long`. To, že 64 bitov obetujeme na jeden bit nevadí, lebo variantný typ je aj tak veľký ako najväčšia položka v ňom. V prípade typu `IS_LONG` je uložené dané číslo, ak ide o typ `IS_BOOL`, tak číslo nadobúda hodnoty nula alebo jedna. Pri type `IS_RESOURCE` je hodnota identifikátorom zdroja (*resource*) v tabuľke zdrojov.

Typ `IS_ARRAY` využíva vo variantnom type člen `HashTable`, `IS_OBJECT` využíva člen `zend_object_value`. Konštanty sú typov `IS_CONSTANT` a `IS_CONSTANT_AST`. Typ `IS_CONSTANT_AST` využíva člen `zend_ast`. Viac o týchto typoch si povieme v častiach 4.2.2, 4.2.5 a 4.2.3.

4.2.2 Hašovacia tabuľka

Polia v jazyku C sú iba oblasti pamäte, na ktoré sa pristupuje pomocou offsetu, teda pristupuje sa pomocou číselných indexov, ktoré musia ísť za sebou. Polia v PHP sú odlišné. Ako kľúč môže byť použité číslo (indexy nemusia ísť za sebou), ale aj reťazce, dokonca môžu byť použité obe zároveň. V jazyku C je to možné implementovať buď ako binárny vyhľadávací strom, kde vloženie nového prvku a test na prítomnosť prvku majú zložitosť $O(\log n)$, kde n je počet prvkov poľa, alebo ako hašovaciu tabuľku, kde zložitosť na test prítomnosti prvku a vloženie nového prvku je $O(1)$. PHP používa na reprezentáciu polí hašovaciu tabuľku.

Myšlienka hašovacej tabuľky je pretransformovať komplexnú dátovú štruktúru, v našom prípade reťazec alebo číslo, na číslo, ktoré môže byť použité ako index do poľa. Môže nastať situácia, kedy sa dve rôzne dátové štruktúry pretransformujú na rovnaké číslo. Táto situácia sa nazýva kolízia. Na riešenie kolízií existujú v podstate dve riešenia. Pri *otvorenom adresovaní* sa prechádza poľom, až kým sa prvok nájde, alebo po prvé voľné miesto. Na prechádzanie poľom sa môžu použiť rôzne techniky, napríklad zvyšovaním o jedna (*linear probing*), pripočítavaním nejakej hodnoty, ktorú

Ukážka kódu 4.3: Bucket v PHP 5

```
typedef struct bucket {
        ulong h;        /* Used for numeric indexing */
        uint nKeyLength;
        void *pData;
        void *pDataPtr;
        struct bucket *pListNext;
        struct bucket *pListLast;
        struct bucket *pNext;
        struct bucket *pLast;
        const char *arKey;
} Bucket;
```

lineárne zvyšujeme (*quadratic probing*) alebo pripočítavaním hodnoty získanej z inej hašovacej funkcie (*double hashing*). Druhým spôsobom, ktorý využíva aj PHP, je *reťazenie*. Prvky s rovnakým hašom sú zaradené do spájaného zoznamu. Pri hľadaní prvku sa následne prechádza týmto spájaným zoznamom, kým nenastane zhoda alebo až po koniec zoznamu.

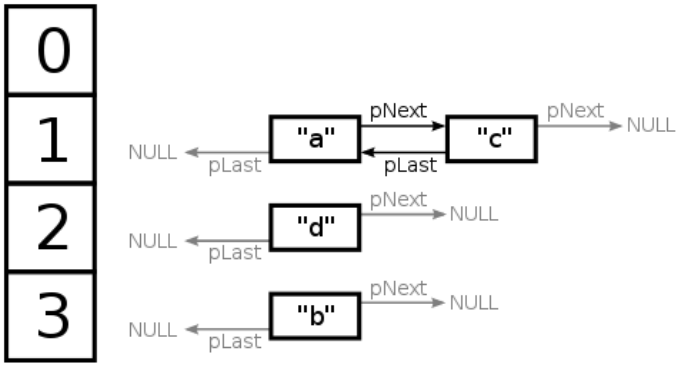
Jednotlivé prvky hašovacej tabuľke nesú pomenovanie `Bucket`, ktorý je znázornený v ukážke kódu 4.3. Index alebo haš kľúča sú uložené v položke `h`. V prípade, že kľúčom je reťazec, jeho dĺžka je uchovaná v člene `nKeyLength` a smerník na reťazec je uložený v položke `arKey`. Na rozdiel od reťazca v štruktúre `zval` je tu reťazec ukončený nulovým bajtom, ktorý je zarátaný aj v dĺžke reťazca. Ak je ako index použité číslo, tak položky `nKeyLength` a `arKey` majú hodnotu nula.

Uchovávanie hodnoty, ktorú `Bucket` obsahuje, zabezpečujú členy `pData` a `pDataPtr`. V prípade, že je uchovávaná štruktúra `zval`, tak `pDataPtr` bude obsahovať smerník na túto štruktúru. Zároveň, `pData` bude smerníkom na `pDataPtr`.

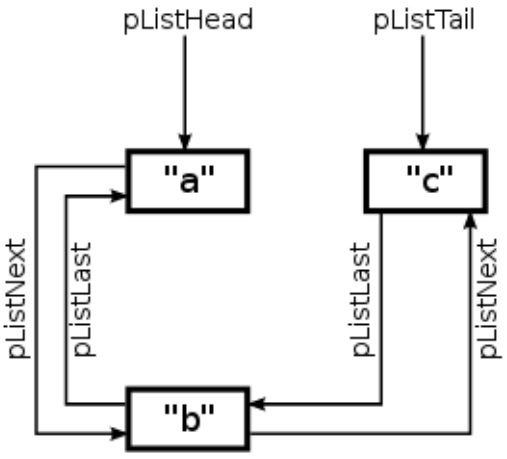
Na riešenie kolízií sa používajú členy `pNext` a `pLast`, ktoré fungujú ako obojsmerný spájaný zoznam a sú znázornené na obrázku 4.1. Položky `pListLast` a `pListNext` nám udávajú predchádzajúci, respektíve nasledujúci `Bucket` vložený do hašovacej tabuľky z globálneho pohľadu. Znázornené sú na obrázku 4.2.

Štruktúra použitá na reprezentáciu hašovacej tabuľky je znázornená v ukážke kódu 4.4. Položka `nTableSize` určuje veľkosť tabuľky. Keďže sa jedná o štruktúru s premenlivým počtom prvkov, nebude tento člen pevne určený. Jeho hodnota je najnižšia mocnina dvojky, taká že počet prvkov tabuľky je menší, nanajvýš rovný tomuto číslu. Toto číslo sa nezmení, ak počet prvkov klesne pod nižšiu mocninu dvojky.

Výsledkom hašovacej funkcie je `ulong`, teda jeho hodnota bude zvyčajne väčšia ako



Obr. 4.1: Riešenie kolízií Bucketov [12]



Obr. 4.2: Spôsob akým PHP zachováva poradie vložených Bucketov (Buckety boli pridané v poradí *a, b, c*) [12]

Ukážka kódu 4.4: Hashtable v PHP 5

```
typedef struct _hashtable {
        uint nTableSize;
        uint nTableMask;
        uint nNumOfElements;
        ulong nNextFreeElement;
        Bucket *pInternalPointer;
                /* Used for element traversal */
        Bucket *pListHead;
        Bucket *pListTail;
        Bucket **arBuckets;
        dtor_func_t pDestructor;
        zend_bool persistent;
        unsigned char nApplyCount;
        zend_bool bApplyProtection;
#if ZEND_DEBUG
        int inconsistent;
#endif
} HashTable;
```

veľkosť tabuľky. Preto nemôže byť haš použitý priamo ako index. Namiesto toho je ako index použitá hodnota `index = hash % nTableSize`, ktorá v dôsledku toho, že `nTableSize` je mocnina dvojky, je ekvivalentná výrazu `index = hash & (nTableSize - 1)`.

Hodnota `nTableSize - 1` sa nazýva maska a je uložená v člene `nTableMask`. Počet prvkov tabuľky je určený v člene `nNumOfElements`. Obsahom položky `nNextFreeElement` je maximálna hodnota číselného indexu použitého v tabuľke zväčšená o jedna. Táto hodnota sa použije, ak sa pridáva prvok do poľa spôsobom `$pole[] = $hodnota;`. Pri iterácii hašovacou tabuľkou je `pInternalPointer` smerníkom na aktuálne spracovávaný prvok. V člene `arBuckets` je uložený smerník na pole začiatkov spájaných zoznamov `Bucket`-ov pre jednotlivé indexy. Smerník na prvý vložený `Bucket` sa nachádza v člene `pListHead`. Podobne, `pListTail` je smerníkom na posledný vložený `Bucket`. Vďaka týmto dvom členom je možné prechádzať prvky hašovacej tabuľky v poradí, v akom boli do tabuľky vložené a aj naopak.

4.2.3 Konštanty

Konštanty sú svojim spôsobom menami pre jednoduché hodnoty. V PHP sú typov `IS_CONSTANT` a `IS_CONSTANT_AST` a sú deklarované ako: `define("ANIMAL", "giraffe");`. Pre typ `IS_CONSTANT` sa nám nepodarilo určiť, ako je konštanta uložená. V prípade, že je konštanta typu `IS_CONSTANT_AST`, obsahom `zend_value` bude smerník do abstraktného syntaktického stromu.

Ukážka kódu 4.5: Konštanty v PHP 5

```
1 typedef struct _zend_constant {
2         zval value;
3         int flags;
4         char *name;
5         uint name_len;
6         int module_number;
7 } zend_constant;
```

Ako sú uložené konštanty v globálnej hašovacej tabuľke konštánt, možno vidieť v ukážke kódu 4.5. Konštanta pozostáva z dvoch hlavných častí: hodnoty a názvu. Hodnota je uložená v atribúte `value` a je typu `zval`. Názov konštanty je určený členmi `name` a `name_len`, kde `name` je reťazec a `name_len` je jeho dĺžka, podobne ako pri reťazcových kľúčoch v hašovacej tabuľke, vrátane ukončovacieho nulového bajtu.

4.2.4 Garbage collection

Ako sme spomenuli vyššie, v štruktúre `zval` sú atribúty slúžiace na garbage collection pri cyklických referenciách. Niektoré štruktúry `zval` sú zabalené v štruktúre `zval_gc_info`, ktorá je znázornená v ukážke kódu 4.6.

Ukážka kódu 4.6: Štruktúra GC info v PHP 5

```
typedef struct _zval_gc_info {
        zval z;
        union {
                gc_root_buffer       *buffered;
                struct _zval_gc_info *next;
        } u;
} zval_gc_info;
```

Táto štruktúra pridáva jeden smerník na začiatok zoznamu štruktúr `zval_gc_info` alebo smerník na ďalšiu štruktúru `zval_gc_info`. Jednou z možností, kedy sa spustí garbage collection, je tá, keď tento zoznam dosiahne určitú dĺžku, ktorá je fixná, ale je ju možné zmeniť.

4.2.5 Objekt

V prípade, že štruktúra `zval` je typu `IS_OBJECT`, obsahom variantného typu `zvalue_value` je dvojica `zend_object_handle` a `zend_object_handlers`, ktorá je znázornená v ukážke kódu ??.

Ukážka kódu 4.7: Štruktúra zend object value v PHP 5

```
typedef struct _zend_object_value {
        zend_object_handle handle;
        const zend_object_handlers *handlers;
} zend_object_value;
```

Druhým členom dvojice je smerník na štruktúru `zend_object_handlerds`, ktorá obsahuje ovládače (*handlers*) objektu a tie definujú správanie objektu, napr. porovnávanie hodnôt. Prvým členom dvojice je identifikátor objektu – index do poľa `_zend_object_store_buckets` v globálnej štruktúre `zend_object_store`, ktorá je znázornená v ukážke kódu 4.8.

Ukážka kódu 4.8: Štruktúra zend object store v PHP 5

```
typedef struct _zend_objects_store {
        zend_object_store_bucket *object_buckets;
        zend_uint top;
```

```
4         zend_uint size;
5         int free_list_head;
6 } zend_objects_store;
```

Zend objects store

Štruktúra `zend_objects_store` sa nachádza v globálnej štruktúre `_zend_executor_globals`, ktorá je znázornená v časti A.1. Štruktúre `_zend_executor_globals` sa budeme venovať v časti 4.4. Obsahom člena `object_buckets` je smerník na pole štruktúr typu `zend_object_store_bucket`. Položka `size` nám hovorí o veľkosti tohto poľa. Číslo nasledujúceho identifikátora (*handle*), ktorý bude použitý, je uložené v člene `top`. Identifikátory sa začínajú na jednotke, aby neexistoval identifikátor s hodnotou 0. Index, na ktorom začína spájaný zoznamu nepoužitých záznamov typu `zend_object_store_bucket`, je uložený v člene `free_list_head`.

Ukážka kódu 4.9: Štruktúra zend object store bucket v PHP 5

```
1  typedef struct _zend_object_store_bucket {
2      zend_bool destructor_called;
3      zend_bool valid;
4      zend_uchar apply_count;
5      union _store_bucket {
6          struct _store_object {
7              void *object;
8              zend_objects_store_dtor_t dtor;
9              zend_objects_free_object_storage_t
                   free_storage;
10             zend_objects_store_clone_t clone;
11             const zend_object_handlers *handlers;
12             zend_uint refcount;
13             gc_root_buffer *buffered;
14         } obj;
15         struct {
16             int next;
17         } free_list;
18     } bucket;
19 } zend_object_store_bucket;
```

Zend object store bucket

V štruktúre `zend_object_store_bucket`, ktorá je znázornená v ukážke kódu 4.9, sa už nachádza smerník na objekt v člene `object` štruktúry `_store_object` vo variantnom type `_store_bucket`. V tomto variantnom type môžu byť dáta interpretované aj ako index na ďalší nepoužitý `zend_object_store_bucket` v poli `object_buckets`. O tom, či vo variantnom type bude uložený index alebo `_store_object`, rozhoduje člen `valid`. Ak je jeho hodnota 1, obsahom bude štruktúra `_store_object`. V prípade, že hodnota bude 0, obsahom bude index na ďalší nepoužitý `zend_object_store_bucket`.

Ukážka kódu 4.10: Štruktúra zend object v PHP 5

```
typedef struct _zend_object {
        zend_class_entry *ce;
        HashTable *properties;
        zval **properties_table;
        HashTable *guards;
            /* protects from __get/__set ... recursion */
} zend_object;
```

Zend object

Vlastnosti (*properties*) objektu v PHP môžu byť deklarované alebo pridané počas behu. Vlastnosti pridané počas behu sú uložené v hašovacej tabuľke `properties`. Táto hašovacia tabuľka mapuje názvy premenných na ich hodnoty. Hodnoty vlastností deklarovaných v triede sú uložené v poli smerníkov na štruktúry `zval`. Smerník na toto pole je obsahom členu `properties_table`. Do tohto poľa sa pristupuje pomocou offsetu.

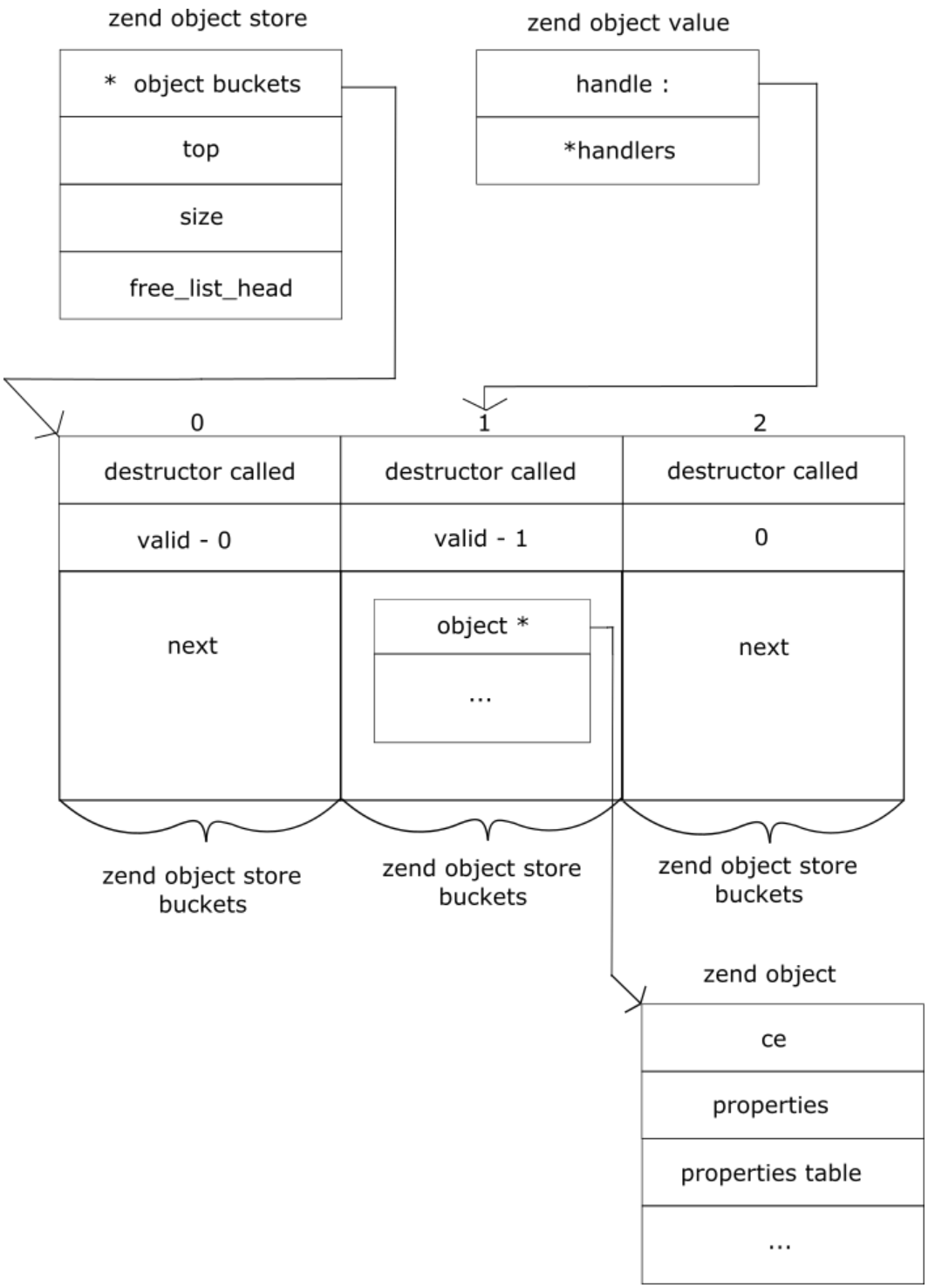
Zend class entry

Na priradenie offsetu k vlastnosti sa používa položka `properties_info` štruktúry `class_entry`. Táto štruktúra obsahuje množstvo informácií, napr. metódy triedy, statické vlastnosti, konštanty triedy, rôzne ovládače... Táto štruktúra je znázornená v ukážke kódu A.3.

Zend property info

Spomínanou štruktúrou na mapovanie mien na offsety je štruktúra `zend_property_info`, ktorá je znázornená v ukážke kódu 4.11

V prípade, že bude objekt obsahovať oba typy vlastností, tak budú použité `properties` a `properties_table` súčasne. Všetky vlastnosti budú uložené v hašovacej tabuľke `properties`, ale `properties_table` bude stále obsahovať smerníky na ne.



Obr. 4.3: Prepojenie štruktúr spojených s typom IS_OBJECT

Ukážka kódu 4.11: Štruktúra zend property info v PHP 5

```
typedef struct _zend_property_info {
        zend_uint flags;
        const char *name;
        int name_length;
        ulong h;
        int offset;
        const char *doc_comment;
        int doc_comment_len;
        zend_class_entry *ce;
} zend_property_info;
```

4.3 PHP 7

V decembri 2015 vydaná verzia PHP 7 je dnes najnovšou stabilnou verziou. Oproti verzii 5 priniesla zrýchlenie, a to aj vďaka novému spôsobu ukladania premenných. Pribudli niektoré štruktúry, iné boli zjednodušené a nejaké zanikli.

4.3.1 Zval

Príkladom je štruktúra `zval`, ktorá je znázornená v ukážke kódu 4.12. Už neobsahuje atribúty používané pre garbage collection. Nahradené boli štruktúrou `zend_refcounted`, o ktorej si povieme v časti 4.3.2.

Ukážka kódu 4.12: Zval v PHP 7

```
typedef struct _zval_struct     zval;

struct _zval_struct {
    zend_value        value;    /* value */
    union {
        struct {
            ZEND_ENDIAN_LOHI_4(
                zend_uchar    type,                    /*
                   active type */
                zend_uchar    type_flags,
                zend_uchar    const_flags,
                zend_uchar    reserved)                /*
                   call info for EX(This) */
        } v;
```

```
13          uint32_t type_info;
14      } u1;
15      union {
16          uint32_t     next;                  /*
                hash collision chain */
17          uint32_t     cache_slot;            /*
                literal cache slot */
18          uint32_t     lineno;                /*
                line number (for ast nodes) */
19          uint32_t     num_args;              /*
                arguments number for EX(This) */
20          uint32_t     fe_pos;                /*
                foreach position */
21          uint32_t     fe_iter_idx;           /*
                foreach iterator index */
22          uint32_t     access_flags;          /*
                class constant access flags */
23          uint32_t     property_guard;        /*
                single property guard */
24          uint32_t     extra;                 /*
                not further specified */
25      } u2;
26  };
```

Variantný typ `zvalue_value` tiež prešiel zmenami, vo verzii sedem sa zmenil názov na `zend_value`. Variantný typ je znázornený v ukážke kódu 4.13. Počet položiek, ktoré môže tento variantný typ reprezentovať, vzrástol. Jediné položky, ktoré obsahujú priamo hodnoty, sú `lval` a `dval`, v ostatných prípadoch ide o smerníky na jednotlivé dátové štruktúry, ktorými sú dané typy reprezentované.

Pribudli aj dátové typy, všetky sú uvedené v tabuľke 4.2 a bližšie si o nich povieme v častiach, v ktorých budú využívané. Člen štruktúry `zval u1` slúži na určenie typu, ktorý je v štruktúre uložený. Makro `ZEND_ENDIAN_LOHI_4` slúži na vyrovnanie sa s endianitou systému. Obsah členov `type_info` a `v` je rovnaký. Dve varianty existujú kvôli lepšej efektivite získavania dát z týchto členov. Keďže veľkosť štruktúr je zarovnávaná na násobok ôsmich a súčet veľkostí členov `value` (8 bajtov) a `u1` (4 bajty) je dvanásť bajtov; zostávajúce štyri bajty sú vyplnené členom `u2`. Člen `u2` je využívaný pri špeciálnych situáciach, napr. pri riešení kolízií v hašovacej tabuľke.

Ukážka kódu 4.13: ariantný typ zend value v PHP 7

```
typedef union _zend_value {
        zend_long         lval;          /* long value */
        double            dval;          /* double value */
        zend_refcounted  *counted;
        zend_string      *str;
        zend_array       *arr;
        zend_object      *obj;
        zend_resource    *res;
        zend_reference   *ref;
        zend_ast_ref     *ast;
        zval             *zv;
        void             *ptr;
        zend_class_entry *ce;
        zend_function    *func;
        struct {
                uint32_t w1;
                uint32_t w2;
        } ww;
} zend_value;
```

Name	Value
IS_UNDEF	0
IS_NULL	1
IS_FALSE	2
IS_TRUE	3
IS_LONG	4
IS_DOUBLE	5
IS_STRING	6
IS_ARRAY	7
IS_OBJECT	8
IS_RESOURCE	9
IS_REFERENCE	10
IS_CONSTANT	11
IS_CONSTANT_AST	12
_IS_BOOL	13
IS_CALLABLE	14
IS_ITERABLE	19
IS_VOID	18
IS_INDIRECT	15
IS_PTR	17
_IS_ERROR	20

Tabuľka 4.2: Dátové typy v PHP 7

Ukážka kódu 4.14: Štruktúra využívaná na počítanie referencií v PHP 7

```
typedef struct _zend_refcounted zend_refcounted;

typedef struct _zend_refcounted_h {
    uint32_t          refcount;                    /*
       reference counter 32-bit */
    union {
        struct {
            ZEND_ENDIAN_LOHI_3(
                zend_uchar    type,
                zend_uchar    flags,                /*
                   used for strings & objects */
                uint16_t      gc_info)              /*
                   keeps GC root number (or 0) and color */
        } v;
        uint32_t type_info;
    } u;
} zend_refcounted_h;

struct _zend_refcounted {
    zend_refcounted_h gc;
};
```

4.3.2 Refcounted

Pre garbage collection bude slúžiť štruktúra `zend_refcounted`. Znázornená je v ukážke kódu 4.14. Jednoduché dátové typy ako `IS_TRUE, IS_FALSE, IS_NULL, IS_UNDEF`, celé (`IS_LONG`) a desatinné (`IS_DOUBLE`) čísla túto štruktúru nevyužívajú. Táto štruktúra nahradila členy `refcount__gc` a `is_ref_gc` štruktúry `zval` verzie 5. Položka `type` sa zhoduje s typom štruktúry `zval`, v ktorej je obsiahnutá.

4.3.3 String

Ukladanie reťazcov už nie je realizované priamo vo variantnom type `zend_value`, ale v samostatnej štruktúre `zend_string`, ktorá je znázornená v ukážke kódu 4.15. Reťazcom prislúcha typ `IS_STRING`. Táto štruktúra obsahuje štruktúru `zend_refcounted` na počítanie referencií. Ďalej je prítomný člen `h`, v ktorom je uložený haš reťazca, nasleduje dĺžka reťazca bez ukončovacieho nulového bajtu. Ďalšia zmena je, že reťazec sa

nachádza priamo v štruktúre v člene `val`, ktorý je síce definovaný ako jednoznakový, ale je posledný v štruktúre, teda pri alokácii dostatočného miesta nenastane prepísanie dát.

Ukážka kódu 4.15: Reťazce v PHP 7

```
typedef struct _zend_string     zend_string;

struct _zend_string {
        zend_refcounted_h gc;
        zend_ulong        h;         /* hash value */
        size_t            len;
        char              val[1];
};
```

4.3.4 Hašovacia tabuľka

Aj v prípade hašovacej tabuľky nastali zmeny, síce je stále typu `IS_ARRAY`, ale štruktúry `Bucket`, ktorá je zobrazená v ukážke kódu 4.16, bola zjednodušená a zmenám sa nevyhla ani štruktúra `HashTable`, ktorá je znázornená v ukážke kódu 4.17.

Ukážka kódu 4.16: Bucket v PHP 7

```
typedef struct _Bucket {
    zval          val;
    zend_ulong    h;   /* hash value (or numeric index)   */
    zend_string *key; /* string key or NULL for numerics */
} Bucket;
```

V štruktúre `Bucket` sa zachoval člen `h`, ktorý obsahuje index, prípadne haš, ak je ako kľúč použitý reťazec. Tu rovnaké veci končia. `Bucket` už neobsahuje smerníky na predchádzajúci, respektíve nasledujúci vložený `Bucket`. Rovnako neobsahuje smerník na ďalší, respektíve predchádzajúci `Bucket` v prípade riešenia kolízií. Ak bol ako kľúč použitý reťazec, člen `key` predstavuje smerník na štruktúru `zend_string`. Keď je použitý číselný kľúč, hodnota tohto atribútu bude nula.

Ukážka kódu 4.17: Hašovacia tabuľka v PHP 7

```
typedef struct _zend_array HashTable;

struct _zend_array {
        zend_refcounted_h gc;
        union {
```

```
6               struct {
7                   ZEND_ENDIAN_LOHI_4(
8                       zend_uchar    flags,
9                       zend_uchar    nApplyCount,
10                      zend_uchar    nIteratorsCount,
11                      zend_uchar    consistency)
12              } v;
13              uint32_t flags;
14          } u;
15          uint32_t          nTableMask;
16          Bucket           *arData;
17          uint32_t          nNumUsed;
18          uint32_t          nNumOfElements;
19          uint32_t          nTableSize;
20          uint32_t          nInternalPointer;
21          zend_long         nNextFreeElement;
22          dtor_func_t       pDestructor;
23  };
```

Rovnaký význam ako vo verzii 5 majú v hašovacej tabuľke členy `nNumOfElements`, `nTableSize`, `nInternalPointer` a `nNextFreeElement`.

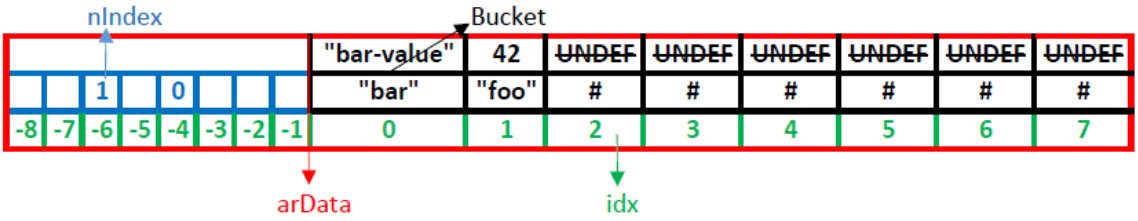
Zmenil sa spôsob akým sú `Bucket`-y uložené v pamäti. Vo verzii 5 boli uložené v obojsmernom spájanom zozname. Teraz sú všetky uložené v jednom poli, ktorého začiatok je obsahom členu `arData`. Toto pole je veľkosti `nTableSize` a prvky sú v takom poradí, v akom boli vložené.

Najnižší voľný index je v člene `nNumUsed`. Ak by sme chceli použiť haš priamo ako index alebo index väčší ako veľkosť tabuľky, mohlo by sa porušiť poradie vložených prvkov v poli.

Ako riešenie je použité pole indexov, ktoré transformuje haš kľúča do indexu, na ktorom sa prvok nachádza. Toto pole indexov sa nachádza hneď pred poľom `arData`. Tieto polia sú spolu alokované ako jeden celok.

Položka `nTableMask` je záporná, aby sa ľahko indexovalo do poľa indexov. Index sa vypočíta ako `index = hash | nTableMask`. Teda hodnoty indexov sú od $-$`nTableMask` do -1. Príklad je znázornený na obrázku 4.4.

V prípade kolízie získame index ďalšieho `Bucket`-u z člena `u2.next` v štruktúre `zval`, ktorý je obsiahnutý v štruktúre `Bucket`. Ak index v poli `arData` nie je použitý, jeho obsahom bude `Bucket`, ktorého `zval` bude typu `IS_UNDEF`. Ak nebude využitý index v poli indexov, bude mať hodnotu `HT_INVALID_IDX`.



Obr. 4.4: Riešenie poradia vložených prvkov s reťazcovým kľúčom v PHP 7, najprv je vložený Bucket s kľúčom „bar“ s hodnotou hašu je 4, potom je pridaný Bucket s kľúčom „foo“ s hodnotou hašu 2

Jednoduché pole

Prvky sú vkladané do `arData` na indexy od 0 až po koniec poľa. Toto nám umožňuje jednoducho iterovať medzi prvkami hašovacej tabuľky. Stačí iterovať poľom `arData`. Ak je použitý ako kľúč reťazec alebo číslo mimo poradia, musíme použiť pole indexov, aby sme mohli pretransformovať haš alebo číslo na index. Existuje ale prípad, kedy je pole indexov úplne zbytočné. Je to vtedy, ak sú použité len číselné kľúče v rastúcom poradí. Pole indexov síce nie je teraz potrebné, ale bude obsahovať dva indexy s hodnotou `HT_INVALID_IDX`. Informácia, či je hašovacia tabuľka v takomto stave, je obsiahnutá v člene `flags` štruktúry `v` variantného typu `u` s hodnotou `HASH_FLAG_PACKED`. Ak by bol teraz vložený prvok s reťazcovým alebo nevhodným číselným kľúčom, je nutné prerobiť hašovaciu tabuľku do formy s platným poľom indexov.

4.3.5 Objekt

Ak je `zval` typu `IS_OBJECT`, tak obsahom variantného typu `zend_value` bude smerník na štruktúru `zend_object`, ktorá je znázornená v ukážke kódu 4.18. Najväčšou zmenou oproti verzii 5 je, že štruktúra `zend_object` obsahuje všetky dáta objektu a netreba k nemu pristupovať cez tabuľku objektov. Táto štruktúra obsahuje štruktúru `zend_refcounted` na počítanie referencií. Význam členov `ce`, `properties` a `properties_table` štruktúry `zend_object` je analogický tým, ktoré sú uvedené vo verzii 5.

Ukážka kódu 4.18: Objekt v PHP 7

```
1 typedef struct _zend_object      zend_object;
2
3 struct _zend_object {
4         zend_refcounted_h gc;
5         uint32_t          handle;                    //
            TODO: may be removed ???
```

```
6         zend_class_entry *ce;
7         const zend_object_handlers *handlers;
8         HashTable         *properties;
9         zval               properties_table[1];
10 };
```

4.3.6 Referencie

Nová je štruktúra `zend_reference`, ktorá slúži na uchovávanie referencií, ktoré fungujú podobne ako v jazyku C, a ktoré sú typu `IS_REFERENCE` (príklad v ukážke kódu 4.19).

Ukážka kódu 4.19: Referencovanie v PHP

```
1                         $a = 5;
2                         $b = & $a;
```

Referencie využívajú štruktúru znázornenú v ukážke kódu 4.20. Táto štruktúra obsahuje štruktúru `zend_refcounted` na počítanie referencií. Samotná hodnota je uložená v člene `val`, ktorý je typu `zval`.

Ukážka kódu 4.20: Referencie v PHP 7

```
1 typedef struct _zend_reference  zend_reference;
2
3 struct _zend_reference {
4         zend_refcounted_h gc;
5         zval              val;
6 };
```

4.3.7 Konštanty

Podobne ako vo verzii 5, aj vo verzii 7 sú pre konštanty použité typy `IS_CONSTANT` a `IS_CONSTANT_AST`. Ak ide o typ `IS_CONSTANT_AST`, obsahom variantného typu `zend_value` bude smerník na štruktúru `zend_ast_ref`, ktorá je znázornená v ukážke kódu 4.21.

Ukážka kódu 4.21: Štruktúra zend ast ref v PHP 7

```
1 typedef struct _zend_ast_ref    zend_ast_ref;
2
3 struct _zend_ast_ref {
4         zend_refcounted_h gc;
5         zend_ast         *ast;
6 };
```

Táto štruktúra obsahuje štruktúru `zend_refcounted` na počítanie referencií a ďalej smerník do abstraktného syntaktického stromu. V prípade typu `IS_CONSTANT` ide o smerník na štruktúru `zend_string`, ktorá obsahuje názov konštanty.

Ukážka kódu 4.22: Konštanty v PHP 7

```
typedef struct _zend_constant {
        zval value;
        zend_string *name;
        int flags;
        int module_number;
} zend_constant;
```

Všetky konštanty sú uložené v globálnej štruktúre `_zend_executor_globals` (popísaná v časti 4.4 a znázornená v ukážke kódu B.1) v hašovacej tabuľke `zend_constants`. Štruktúra `zval` v štruktúre `Bucket` bude typu `IS_PTR` a ako hodnotu bude obsahovať smerník na štruktúru `zend_constant`, ktorá je zobrazená v ukážke kódu 4.22. Hodnota konštanty je uložená v člene `value`, ktorý je typu `zval`. Názov konštanty je uložený v člene `name`, ktorý je typu `zend_string`.

4.3.8 Resource

Zdrojom prislúcha typ `IS_RESOURCE`. Ako sú zdroje uložené v pamäti, možno vidieť v ukážke kódu 4.23. Na počítanie referencií slúži štruktúra `zend_refcounted` v položke `gc`. Typ zdroju je určený členom `type`. Smerník na dáta zdroju je uložený v položke `ptr`. Dáta samotných zdrojov už nemajú ustálený formát.

Ukážka kódu 4.23: Resourci v PHP 7

```
typedef struct _zend_resource   zend_resource;

struct _zend_resource {
        zend_refcounted_h gc;
        int               handle;                    //
           TODO: may be removed ???
        int               type;
        void             *ptr;
};
```

4.4 Iné významné štruktúry

Okrem už spomenutých štruktúr, PHP obsahuje aj niekoľko dôležitých globálnych štruktúr. Väčšina z nich sú buď typu *compiler_globals struct* alebo *executor_globals struct* a najčastejšie sú referencované makrami *CG()* a *EG()*. Niektoré z nich si teraz vymenujeme. Štruktúry sú znázornené v ukážkach kódu A.1 a A.2 pre verziu 5, respektíve B.1 a B.2 pre verziu 7.

- ***CG(function_table)* a *EG(function_table)*** sú hašovacie tabuľky, v ktorých sú uchované všetky funkcie,
- ***CG(class_table)* a *EG(class_table)*** sú hašovacie tabuľky, v ktorých sú uchované všetky triedy,
- ***EG(symbol_table)*** je hašovacia tabuľka, ktorá je hlavnou tabuľkou symbolov a sú tu uložené všetky premenné z globálneho pohľadu vykonávania (*global scope*),
- ***EG(active_symbol_table)*** je hašovacia tabuľka, ktorá obsahuje tabuľku symbolov z aktuálneho pohľadu vykonávania (*current scope*),
- ***EG(zend_constants)*** je hašovacia tabuľka, v ktorej sú konštanty definované funkciou `define`,
- ***CG(auto_globals)*** je hašovacia tabuľka superglobálnych premenných (*superglobals*), ktoré sú použité v skripte. Keďže sú typu *compiler_globals*, tak sú inicializované, iba ak ich skript používa. Medzi superglobálne premenné patria *$GLOBALS*, *$_SERVER*, *$_GET*, *$_POST*, *$_FILES*, *$_COOKIE*, *$_SESSION*, *$_REQUEST*, *$_ENV*.
- ***EG(regular_list)*** je hašovacia tabuľka, ktorá obsahuje zdroje, ktoré nie sú perzistentné,
- ***EG(persistent_list)*** je podobne ako *EG(regular_list)* hašovacia tabuľka, ale *EG(persistent_list)* obsahuje zdroje, ktoré sú perzistentné (nie sú uvoľňované na konci každej požiadavky), napr. perzistentné spojenia s databázou.

Vymenované štruktúry platia pre PHP verziu 5. Vo verzii 7 bola hašovacia tabuľka `active_symbol_table` presunutá do štruktúry `zend_execute_data`.

Kapitola 5

Implementácia

V tejto kapitole popíšeme niektoré implementačné detaily aplikácie, jej nedostatky a možnosti rozšírenia pre vybrané súborov.

5.1 Python-haystack

Pôvodne mala naša aplikácia využívať framework python-haystack. Ide o framework na analýzu haldy procesu (*process heap*) zameraný na hľadanie a reverzovanie štruktúr jazyka C alokovaných v pamäti [6].

Prvá skupina funkcií, ktorú tento framework poskytuje, slúži na hľadanie známych záznamov v obraze procesu. Druhá skupina funkcií je určená pre reverzné inžinierstvo pamäťových záznamov prítomných v halde procesu. Zameriava sa na rekonštrukciu a klasifikáciu klasických štruktúr jazyka C z pamäte a pokúša sa obnoviť typové definície.

Definovanie vlastných štruktúr je možné pomocou knižnice ctypes, ktorá poskytuje C kompatibilné dátové typy a umožňuje volanie funkcií z dynamicky linkovaných (DLL) alebo zdieľaných knižníc. Na jednotlivých členoch definovaných štruktúr môže vykonávať kontroly: člen je nenulový, člen je z nejakej množiny hodnôt, obsahom členu je valídny smerník, rekurzívna kontrola takejto štruktúry a aj ignorovanie člena.

Dôvod, prečo sme sa rozhodli nepoužiť tento framework, je, že potrebujeme kontrolovať aj hodnoty medzi členmi jednej štruktúry (napríklad vzťah medzi maskou a veľkosťou hašovacej tabuľky). Framework síce poskytuje možnosť doplnenia kontrol na členy, ale implementácia by bola zložitejšia ako vytvorenie vlastného spôsobu kontroly. Preto tento framework nebol v našej aplikácii použitý.

5.2 Aplikácia

Aplikácia ponúka tri spôsoby hľadania premenných. Prvým z nich je prechádzanie všetkých adries a na každej vytváranie štruktúry `zval`. Týmto spôsobom získame všetky

obsahy premenných, ktoré sú definované v skripte a existujú v momente vytvorenia obrazu pamäti. Druhým spôsobom je priame vytváranie typov na všetkých adresách. Vďaka tomuto spôsobu sme schopný získať všetky hodnoty, ktoré by sme obdržali využitím minulého spôsobu, ale aj mnoho ďalších. Príčinou je, že mnoho vnútorných štruktúr využíva hašovaciu tabuľku, ktorá ale nie je obalená štruktúrou `zval`. Tento spôsob má aj svoju nevýhodu. Je ňou veľké množstvo falošných pozitívnych nálezov.

Čím zložitejšia štruktúra, tým menej bolo falošných pozitív a interné štruktúry boli rozoznateľné od tých, ktoré by mohli byť definované v skripte. Už pri použití prvého spôsobu bolo toto číslo veľké. Pri niektorých typoch, hlavne vo verzii sedem sa nedali použiť žiadne obmedzenia, napr. typy `IS_UNDEF`, `IS_NULL`, `IS_LONG`, `IS_DOUBLE`, `IS_TRUE` a `IS_FALSE`. Rozhodnutie či bude tento typ valídny záviselo len na jednom bajte v štruktúre `zval`, ktorá určila o aký typ ide. Podobne vo verzii päť je možná jediná kontrola a to pri určení typu. Vyskytuje sa napríklad pri typoch `IS_NULL`, `IS_LONG` a `IS_DOUBLE`. V prípade použitia druhého spôsobu, kde sa vytvárajú len samotné typy, nebude možná pri vymenovaných typoch ani táto jedna kontrola.

Tretím a posledným spôsobom, ktorý mal výrazne zredukovať množstvo falošných pozitív, bolo nájdenie globálnej štruktúry `_zend_executor_globals`. Z nej sme prechádzaním zoznamu vykonávaných funkcií a získavaním ich tabuľky symbolov mali získať len tie obsahy premenných, ktoré v tom momente v skripte existujú a mali mať k nim priradené aj mená. Z neznámych dôvodov, pri prechádzaní zoznamu volaní, nepoužívala žiadna funkcia, okrem hlavného tela skriptu, teda jej premenné boli globálne. Následne sme vyskúšali vytvoriť na každej adrese hašovaciu tabuľku. Ani vo výstupe sa nenachádzali žiadne tabuľky, ktoré by mohli byť tabuľkami symbolov pre jednotlivé funkcie. Teda premenné v rámci funkcie sú zrejme uložené v nejakom poli a v inom poli k nim prislúchajúce názvy.

Tretí spôsob sa vo verzii 5 bude využívať aj pri vypisovaní premenných prvými dvomi spôsobmi. Dôvodom je potreba štruktúry `_zend_object_store` pri mapovaní indexov do tejto tabuľky na jednotlivé objekty. Pri vypísaní štruktúry `_zend_executor_globals`, sú vypísané aj tabuľka tried, funkcii a konštánt. Tieto tabuľky obsahujú veľké množstvo interných tried, funkcií a konštánt PHP. PHP si udržuje aj poradie vložených prvkov do hašovacej tabuľky z dôvodu použitia funkcie `foreach`. Preto triedy, funkcie a konštanty definované v skripte sa budú nachádzať na koncoch svojich zoznamov. Naša aplikácia vypisuje prvky hašovacej tabuľky v takom poradí v akom boli vložené. Vo verzii 5 sme využili člen hašovacej tabuľky `pListHead` a člen štruktúry `Bucket pListNext`. Pri verzii 7 sme prešli cez pole `arData` a získavali platné štruktúry `zval`. Z neznámych dôvodov, pri niektorých hašovacích tabuľkách vo verzii sedem sa niektoré hodnoty vyskytovali viackrát, alebo úplne nesúvisiace hodnoty, teda počet platných prvkov tabuľky bol väčší ako počet deklarovaných prvkov v hlavičke hašovacej tabuľky. Je to zvláštne aj z dôvodu, že presne takto funguje funkcia `foreach` vo verzii

7 na hašovacej tabuľke. Teda ak by bola zavolaná, mala by vrátiť aj tieto hodnoty, ktoré by tam vôbec nemali byť.

Konštanty sa nám podarilo nájsť iba v tabuľke konštánt.

Na parsovanie súboru ELF sme použili knižnicu pyelftools.

Zrýchlenie programu

Naša aplikácia potrebovala na počítači s procesorom Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz a 4 GB RAM na spracovanie 6 MB súboru 30 sekúnd (čas je platný pre verziu sedem s vytváraním štruktúr typu `zval` na všetkých adresách, okrem štruktúr `zval` typu `IS_OBJECT`). Tieto hodnoty sú pre reálne obrazy pamäte nevyhovujúce, keďže môžu mať veľkosť až niekoľko stoviek megabajtov. Z uvedeného dôvodu je nutné program optimalizovať. Preto teraz popíšeme niektoré spôsoby na zrýchlenie aplikácie (jeden z nich je aj implementovaný).

Jedným z opatrení na zrýchlenie programu môže byť zredukovanie segmentov na také, do ktorých je možné zapisovať. V testovacích súboroch sme vďaka tomuto znížili počet segmentov o viac ako polovicu. Tento krok ale môže spôsobiť aj zneplatnenie niektorých štruktúr v prípade, ak by niektoré štruktúry obsahovali smerníky do nepoužitých segmentov (napríklad funkcie).

Zmenený bol spôsob akým získaval program dáta zo súboru. V pôvodnej verzii mala trieda jeden parameter konštruktora identifikátor analyzovaného súboru. Vždy, keď potreboval dáta zo súboru, nastavil sa na správnu pozíciu v súbore a prečítal potrebné množstvo bajtov. Vyriešené to bolo pomocou globálnej premennej, do ktorej sa pri volaní funkcie `init` načíta celý obsah súboru. Keď program potrebuje dáta zo súboru, zavolá globálnu funkciu `get_data`, ktorá vráti potrebné bajty. Problémom môžu byť následne veľké súbory.

Iným spôsobom by bolo použitie už vytvorených dátových štruktúr. Ak teda program vytvára na adrese x nejakú štruktúru a na tejto adrese už takáto štruktúra existovala, tak by sa použila už existujúca štruktúra. Toto vylepšenie by zároveň vyriešilo problém, kedy „správne bajty“ spôsobia cyklické vytváranie štruktúry na adrese.

Iným spôsobom na zrýchlenie, je upravenie funkcie `valid_address`. Táto funkcia prehľadáva pole segmentov a zisťuje či je adresa vrámci hraníc segmentu, avšak takmer vždy, keď táto funkcia vráti `True`, je nasledovaná volaním funkcie `get_offset`, ktorá znova prehľadáva pole segmentov. Riešením by mohlo byť upravenie funkcie `valid_address` tak, aby vracala offset adresy v súbore alebo -1, ak je adresa neplatná.

Neefektívny je aj spôsob vyhľadávania v poli segmentov. Toto pole je prechádzané lineárne. Ak by boli segmenty v poli usporiadané v rastúcom poradí, bolo by možné použiť binárne prehľadávanie.

S funkciou `valid_address` súvisí aj ďalšie možné zrýchlenie. Na mnohých miestach,

kde sa vytvára nejaký typ a parametrom je smerník na štruktúru, ktorú sa snažíme vytvoriť, sa vykoná kontrola adresy pred vytvorením štruktúry ale aj v jej konštruktore. Riešením by bolo zavedenie konvencie, či sa budú adresy kontrolovať pred vytvorením štruktúry alebo v jej konštruktore.

V prípade ak hľadáme štruktúru `_zend_executor_globals`, rýchlosť aplikácie pre verziu 7 je oveľa vyššia ako pre verziu 5. Súvisí to s množstvom kontrol na zistenie platnosti štruktúry. V prípade verzie 5 ide o 14 položiek, ktoré kontrolujeme. 12 z nich je testom na platnosť adresy, zvyšné dve sa pokúsia vytvoriť hlavičku hašovacej tabuľky a overiť jej platnosť. Pri verzii sedem bolo použitých 6 položiek na kontrolu. Jedenkrát sa testovala platnosť adresy, dvakrát vytvorenie hašovacej tabuľky na adrese a trikrát vytvorenie hašovacej tabuľky zo smerníku na ňu. Aj napriek značne menšiemu počtu kontrol oproti verzii 5 sme nezaznamenali na testovacích vstupoch žiadne falošné pozitíva. Preto by sa mohol výrazne znížiť počet kontrol pre verziu 5.

Zrýchlenie by priniesla aj zmena programovacieho jazyka, ak by bol namiesto Pythonu použitý napríklad jazyk C/C++.

5.3 Rozšírenia a návrhy na zlepšenie

Rozšírenia

Pre rozšírenie aplikácie na ľubovoľný súborový formát stačí v súbore ?? upraviť funkciu `get_segments` tak, aby vracala mapovanie segmentov na offsety v súbore vo formáte zoznamov slovníkov, ktoré obsahujú atribúty uvedené v ukážke kódu 5.1.

Ukážka kódu 5.1: Atribúty, ktoré musí slovník obsahovať pre správne mapovanie segmentov na offsety v súbore

```
{
"file_addr_begin": %offset of first virtual address of
   segment in file%,
"file_addr_end":   %offset of last virtual address + 1 of
   segment in file%,
"virt_addr_begin": %first virtual address of segment%,
"virt_addr_end":   %last virtual address of segment + 1%
}
```

Obmedzením je ale nutnosť, aby bol súbor 64-bitový, lebo veľkosti premenných sú uspôsobené na 64-bitový systém. Problémom je aj tabuľka perzistentných a neperzistentných zdrojov pri systéme Windows, lebo pred nimi sa v štruktúre `_zend_executor_globals` nachádza ešte štruktúra `OSVERSIONINFOEX` v prípade verzie sedem. Vo verzii päť je navyše prítomný aj člen `timed_out` typu `zend_bool`. Na vyriešenie je potrebné v

súboroch *php7/compiler_ executor_ globals.py* a *php5/executor_ globals.py* zväčšiť offset hašovacích tabuliek `regular_list` a `persistent_list` o príslušné hodnoty.

Možným rozšírením aplikácie je pridanie vypisovania rodičovských tried. Momentálne aplikácia vypisuje len aktuálnu triedu objektu. Funkcie by mohli byť rozšírené o svoj protoyp a zoznam argumentov.

V implementácii sme pre jednoduché dátové typy dostávali veľké množstvo falošných pozitív. Jedným spôsobom ako toto číslo znížiť, by bolo treba upraviť PHP interpreter tak, aby pri alokácii pamäte bola vynulovaná. Tým by nepoužité hodnoty, napr. vo variantnom type ak sa používa položka menšia ako veľkosť variantného typu, boli nula a vedeli by sme pridať ďalšiu podmienku na kontrolu.

Záver

V tejto práci sme popísali spôsob ukladania premenných v interpretovanom programovacom jazyku PHP vo verzii 5 a 7. Popísali sme aj princíp fungovania interpretera jazyka PHP. Vysvetlili sme pojem forenzná analýza pamäte. Nadobudnuté poznatky o spôsobe ukladania premenných sme využili na implementovanie aplikácie, ktorá vypíše z obrazu procesu v pamäti hodnoty premenných.

Aplikácia vypisuje hodnoty všetkých premenných, ale zároveň vypisuje aj veľké množstvo „hodnôt premenných“, ktoré sú vnútornými premennými interpretra, ale aj veľké množstvo falošných pozitív. Čím zložitejšia štruktúra, tým menej bolo falošných pozitív a interné štruktúry boli rozoznateľné od tých, ktoré by mohli byť definované v skripte. Pri niektorých typoch, hlavne vo verzii sedem sa nedali použiť žiadne obmedzenia, napr. typy `IS_UNDEF`, `IS_NULL`, `IS_LONG`, `IS_DOUBLE`, `IS_TRUE` a `IS_FALSE`. Rozhodnutie či bude tento typ valídny záviselo len na jednom bajte v štruktúre `zval`, ktorá určila o aký typ ide. Podobne vo verzii päť je možná jediná kontrola, a to pri určení typu. Vyskytuje sa napríklad pri typoch `IS_NULL`, `IS_LONG` a `IS_DOUBLE`.

Na vyriešenie týchto problémov sme použili globálnu štruktúru `_zend_executor_globals`. Konkrétne sme z nej použili členy reprezentujúce globálnu tabuľku symbolov, tabuľku funkcií a tabuľku konštánt. Vďaka tomu sme získali všetky názvy globálnych premenných aj so správne priradenými hodnotami, ktoré sa nachádzajú v skripte, ale aj interné premenné interpretra. Používateľ znalý PHP už vie rozpoznať, premenné deklarované v skripte a tie, ktoré sú interné. Podobná je situácia v prípade tabuľky tried a konštánt.

Na získanie tabuliek symbolov pre vykonávané funkcie sme použili štruktúru `_zend_execute_data`, ktorá obsahuje informácie o práve vykonávanej funkcii, z ktorých sme využili meno funkcie a tabuľku symbolov. Ďalej táto štruktúra obsahuje smerník na takúto štruktúru, teda informácie funkcii, z ktorej bola aktuálna funkcia zavolaná. Takto sme vytvoril zoznam volaných funkcií s ich menami a tabuľkami symbolov. Z nejakého nám neznámeho dôvodu túto tabuľku symbolov nepoužívali a premenné ukladali niekde inde.

Popísali sme aj súborový formát ELF, keďže naša aplikácia berie ako vstup súbor s obrazom procesu v pamäti.

Literatúra

[1] Charles University and Microsoft. Phalanger. https://phalanger.codeplex.com/.

[2] Facebook. HHVM. http://hhvm.com/.

[3] Facebook. HipHop for PHP. https://www.facebook.com/notes/facebook-engineering/hiphop-for-php-move-fast/280583813919/.

[4] The Volatility Foundation. About the volatility foundation. http://www.volatilityfoundation.org/about.

[5] The Volatility Foundation. An advanced memory forensics framework. https://github.com/volatilityfoundation/volatility.

[6] Loic Jaquemet. python-haystack. https://github.com/trolldbois/python-haystack\left.

[7] Rasmus Lerdorf et al. PHP. http://php.net.

[8] Rasmus Lerdorf et al. The PHP Interpreter. https://github.com/php/php-src.

[9] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 1st edition, 2014.

[10] Parrot Foundation. Parrot. http://parrot.org/.

[11] Julien Pauli. Julien Pauli PHP's life. http://jpauli.github.io/.

[12] Julien Pauli, Anthony Ferrara, and Nikita Popov. PHP Internals Book. http://www.phpinternalsbook.com.

[13] Nikita Popov. Blog by nikic. https://nikic.github.io/aboutMe.html.

[14] GNU Project. gcore. http://man7.org/linux/man-pages/man1/gcore.1.html.

[15] Mark Russinovich. ProcDump. https://phalanger.codeplex.com/.

[16] George Schlossnagle. *Advanced PHP Programming.* Sams, Indianapolis, IN, USA, 2003.

[17] Tool Interface Standards (TIS) Committee. *Executable and Linking Format (ELF)*, May 1995. `http://refspecs.linuxbase.org/elf/elf.pdf`.

[18] W3Techs. Usage statistics and market share of PHP for websites. Získané dňa 16.5.2017 `https://w3techs.com/technologies/details/pl-php/all/all`.

Dodatok A

Štruktúry v PHP 5

Ukážka kódu A.1: Štruktúra executor globals v PHP 5

```
struct _zend_executor_globals {
    zval **return_value_ptr_ptr;

    zval uninitialized_zval;
    zval *uninitialized_zval_ptr;

    zval error_zval;
    zval *error_zval_ptr;

    /* symbol table cache */
    HashTable *symtable_cache[SYMTABLE_CACHE_SIZE];
    HashTable **symtable_cache_limit;
    HashTable **symtable_cache_ptr;

    zend_op **opline_ptr;

    HashTable *active_symbol_table;
    HashTable symbol_table;    /* main symbol table */

    HashTable included_files;  /* files already included */

    JMP_BUF *bailout;

    int error_reporting;
    int orig_error_reporting;
    int exit_status;
```

```
27
28      zend_op_array *active_op_array;
29
30      HashTable *function_table;  /* function symbol table */
31      HashTable *class_table;     /* class table */
32      HashTable *zend_constants;  /* constants table */
33
34      zend_class_entry *scope;
35      zend_class_entry *called_scope;
36          /* Scope of the calling class */
37
38      zval *This;
39
40      long precision;
41
42      int ticks_count;
43
44      zend_bool in_execution;
45      HashTable *in_autoload;
46      zend_function *autoload_func;
47      zend_bool full_tables_cleanup;
48
49      /* for extended information support */
50      zend_bool no_extensions;
51
52      #ifdef ZEND_WIN32
53      zend_bool timed_out;
54      OSVERSIONINFOEX windows_version_info;
55      #endif
56
57      HashTable regular_list;
58      HashTable persistent_list;
59
60      zend_vm_stack argument_stack;
61
62      int user_error_handler_error_reporting;
63      zval *user_error_handler;
64      zval *user_exception_handler;
65      zend_stack user_error_handlers_error_reporting;
```

```
66      zend_ptr_stack user_error_handlers;
67      zend_ptr_stack user_exception_handlers;
68
69      zend_error_handling_t  error_handling;
70      zend_class_entry      *exception_class;
71
72      /* timeout support */
73      int timeout_seconds;
74
75      int lambda_count;
76
77      HashTable *ini_directives;
78      HashTable *modified_ini_directives;
79      zend_ini_entry *error_reporting_ini_entry;
80
81      zend_objects_store objects_store;
82      zval *exception, *prev_exception;
83      zend_op *opline_before_exception;
84      zend_op exception_op[3];
85
86      struct _zend_execute_data *current_execute_data;
87
88      struct _zend_module_entry *current_module;
89
90      zend_property_info std_property_info;
91
92      zend_bool active;
93
94      zend_op *start_op;
95
96      void *saved_fpu_cw_ptr;
97      #if XPFPA_HAVE_CW
98      XPFPA_CW_DATATYPE saved_fpu_cw;
99      #endif
100
101     void *reserved[ZEND_MAX_RESERVED_RESOURCES];
102 };
```

Ukážka kódu A.2: Štruktúra compiler globals v PHP 5

```
struct _zend_compiler_globals {
    zend_stack bp_stack;
    zend_stack switch_cond_stack;
    zend_stack foreach_copy_stack;
    zend_stack object_stack;
    zend_stack declare_stack;

    zend_class_entry *active_class_entry;

    /* variables for list() compilation */
    zend_llist list_llist;
    zend_llist dimension_llist;
    zend_stack list_stack;

    zend_stack function_call_stack;

    char *compiled_filename;

    int zend_lineno;

    zend_op_array *active_op_array;

    HashTable *function_table;  /* function symbol table */
    HashTable *class_table;     /* class table */

    HashTable filenames_table;

    HashTable *auto_globals;

    zend_bool parse_error;
    zend_bool in_compilation;
    zend_bool short_tags;
    zend_bool asp_tags;

    zend_declarables declarables;

    zend_bool unclean_shutdown;

```

```
39      zend_bool ini_parser_unbuffered_errors;
40
41      zend_llist open_files;
42
43      long catch_begin;
44
45      struct _zend_ini_parser_param *ini_parser_param;
46
47      int interactive;
48
49      zend_uint start_lineno;
50      zend_bool increment_lineno;
51
52      znode implementing_class;
53
54      zend_uint access_type;
55
56      char *doc_comment;
57      zend_uint doc_comment_len;
58
59      zend_uint compiler_options;
60          /* set of ZEND_COMPILE_* constants */
61
62      zval      *current_namespace;
63      HashTable *current_import;
64      HashTable *current_import_function;
65      HashTable *current_import_const;
66      zend_bool  in_namespace;
67      zend_bool  has_bracketed_namespaces;
68
69      HashTable const_filenames;
70
71      zend_compiler_context context;
72      zend_stack context_stack;
73
74      /* interned strings */
75      char *interned_strings_start;
76      char *interned_strings_end;
77      char *interned_strings_top;
```

```
78      char *interned_strings_snapshot_top;
79  #ifndef ZTS
80      char *interned_empty_string;
81  #endif
82
83      HashTable interned_strings;
84
85      const zend_encoding **script_encoding_list;
86      size_t script_encoding_list_size;
87      zend_bool multibyte;
88      zend_bool detect_unicode;
89      zend_bool encoding_declared;
90
91  #ifdef ZTS
92      zval ***static_members_table;
93      int last_static_member;
94  #endif
95  };
```

Ukážka kódu A.3: Štruktúra class entry v PHP 5

```
1   typedef struct _zend_class_entry zend_class_entry;
2
3   struct _zend_class_entry {
4       char type;
5       const char *name;
6       zend_uint name_length;
7       struct _zend_class_entry *parent;
8       int refcount;
9       zend_uint ce_flags;
10
11      HashTable function_table;
12      HashTable properties_info;
13      zval **default_properties_table;
14      zval **default_static_members_table;
15      zval **static_members_table;
16      HashTable constants_table;
17      int default_properties_count;
18      int default_static_members_count;
19
```

```
20      union _zend_function *constructor;
21      union _zend_function *destructor;
22      union _zend_function *clone;
23      union _zend_function *__get;
24      union _zend_function *__set;
25      union _zend_function *__unset;
26      union _zend_function *__isset;
27      union _zend_function *__call;
28      union _zend_function *__callstatic;
29      union _zend_function *__tostring;
30      union _zend_function *__debugInfo;
31      union _zend_function *serialize_func;
32      union _zend_function *unserialize_func;
33
34      zend_class_iterator_funcs iterator_funcs;
35
36      /* handlers */
37      zend_object_value (*create_object)(zend_class_entry
          *class_type TSRMLS_DC);
38      zend_object_iterator *(*get_iterator)(zend_class_entry
          *ce, zval *object, int by_ref TSRMLS_DC);
39      int (*interface_gets_implemented)(zend_class_entry
          *iface, zend_class_entry *class_type TSRMLS_DC);
              /* a class implements this interface */
40      union _zend_function
          *(*get_static_method)(zend_class_entry *ce, char*
          method, int method_len TSRMLS_DC);
41
42      /* serializer callbacks */
43      int (*serialize)(zval *object, unsigned char **buffer,
          zend_uint *buf_len, zend_serialize_data *data
          TSRMLS_DC);
44      int (*unserialize)(zval **object, zend_class_entry
          *ce, const unsigned char *buf, zend_uint buf_len,
          zend_unserialize_data *data TSRMLS_DC);
45
46      zend_class_entry **interfaces;
47      zend_uint num_interfaces;
48
```

```
49      zend_class_entry **traits;
50      zend_uint num_traits;
51      zend_trait_alias **trait_aliases;
52      zend_trait_precedence **trait_precedences;
53
54      union {
55          struct {
56              const char *filename;
57              zend_uint line_start;
58              zend_uint line_end;
59              const char *doc_comment;
60              zend_uint doc_comment_len;
61          } user;
62          struct {
63              const struct _zend_function_entry
                    *builtin_functions;
64              struct _zend_module_entry *module;
65          } internal;
66      } info;
67  };
```

Ukážka kódu A.4: Štruktúra zend function v PHP 5

```
1   typedef union _zend_function {
2           zend_uchar type;
3               /* MUST be the first element of this struct! */
4
5           struct {
6                   zend_uchar type;        /* never used */
7                   const char *function_name;
8                   zend_class_entry *scope;
9                   zend_uint fn_flags;
10                  union _zend_function *prototype;
11                  zend_uint num_args;
12                  zend_uint required_num_args;
13                  zend_arg_info *arg_info;
14          } common;
15
16          zend_op_array op_array;
17          zend_internal_function internal_function;
```

```
18 } zend_function;
```

Ukážka kódu A.5: Štruktúra zend execute data v PHP 5

```
1  struct _zend_execute_data {
2      struct _zend_op *opline;
3      zend_function_state function_state;
4      zend_op_array *op_array;
5      zval *object;
6      HashTable *symbol_table;
7      struct _zend_execute_data *prev_execute_data;
8      zval *old_error_reporting;
9      zend_bool nested;
10     zval **original_return_value;
11     zend_class_entry *current_scope;
12     zend_class_entry *current_called_scope;
13     zval *current_this;
14     struct _zend_op *fast_ret;
15         /* used by FAST_CALL/FAST_RET (finally keyword) */
16     zval *delayed_exception;
17     call_slot *call_slots;
18     call_slot *call;
19 };
```

Dodatok B

Štruktúry v PHP 7

Ukážka kódu B.1: Štruktúra executor globals v PHP 7

```
struct _zend_executor_globals {
        zval uninitialized_zval;
        zval error_zval;

        /* symbol table cache */
        zend_array *symtable_cache[SYMTABLE_CACHE_SIZE];
        zend_array **symtable_cache_limit;
        zend_array **symtable_cache_ptr;

        zend_array symbol_table;   /* main symbol table */

        HashTable included_files;                       /*
           files already included */

        JMP_BUF *bailout;

        int error_reporting;
        int exit_status;

        HashTable *function_table;                      /*
           function symbol table */
        HashTable *class_table;    /* class table */
        HashTable *zend_constants; /* constants table */

        zval          *vm_stack_top;
        zval          *vm_stack_end;
```

```
25          zend_vm_stack  vm_stack;
26
27          struct _zend_execute_data *current_execute_data;
28          zend_class_entry *fake_scope;                    /*
              used to avoid checks accessing properties */
29
30          zend_long precision;
31
32          int ticks_count;
33
34          HashTable *in_autoload;
35          zend_function *autoload_func;
36          zend_bool full_tables_cleanup;
37
38          /* for extended information support */
39          zend_bool no_extensions;
40
41          zend_bool vm_interrupt;
42          zend_bool timed_out;
43          zend_long hard_timeout;
44
45  #ifdef ZEND_WIN32
46          OSVERSIONINFOEX windows_version_info;
47  #endif
48
49          HashTable regular_list;
50          HashTable persistent_list;
51
52          int user_error_handler_error_reporting;
53          zval user_error_handler;
54          zval user_exception_handler;
55          zend_stack user_error_handlers_error_reporting;
56          zend_stack user_error_handlers;
57          zend_stack user_exception_handlers;
58
59          zend_error_handling_t  error_handling;
60          zend_class_entry      *exception_class;
61
62          /* timeout support */
```

```
63          zend_long timeout_seconds;
64
65          int lambda_count;
66
67          HashTable *ini_directives;
68          HashTable *modified_ini_directives;
69          zend_ini_entry *error_reporting_ini_entry;
70
71          zend_objects_store objects_store;
72          zend_object *exception, *prev_exception;
73          const zend_op *opline_before_exception;
74          zend_op exception_op[3];
75
76          struct _zend_module_entry *current_module;
77
78          zend_bool active;
79          zend_bool valid_symbol_table;
80
81          zend_long assertions;
82
83          uint32_t           ht_iterators_count;     /*
              number of allocatd slots */
84          uint32_t           ht_iterators_used;      /*
              number of used slots */
85          HashTableIterator *ht_iterators;
86          HashTableIterator  ht_iterators_slots[16];
87
88          void *saved_fpu_cw_ptr;
89  #if XPFPA_HAVE_CW
90          XPFPA_CW_DATATYPE saved_fpu_cw;
91  #endif
92
93          zend_function trampoline;
94          zend_op       call_trampoline_op;
95
96          void *reserved[ZEND_MAX_RESERVED_RESOURCES];
97  };
```

Ukážka kódu B.2: Štruktúra compiler globals v PHP 7

```
struct _zend_compiler_globals {
        zend_stack loop_var_stack;

        zend_class_entry *active_class_entry;

        zend_string *compiled_filename;

        int zend_lineno;

        zend_op_array *active_op_array;

        HashTable *function_table;                      /*
           function symbol table */
        HashTable *class_table;         /* class table */

        HashTable filenames_table;

        HashTable *auto_globals;

        zend_bool parse_error;
        zend_bool in_compilation;
        zend_bool short_tags;

        zend_bool unclean_shutdown;

        zend_bool ini_parser_unbuffered_errors;

        zend_llist open_files;

        struct _zend_ini_parser_param *ini_parser_param;

        uint32_t start_lineno;
        zend_bool increment_lineno;

        zend_string *doc_comment;
        uint32_t extra_fn_flags;

        uint32_t compiler_options;                      /*
```

```
            set of ZEND_COMPILE_* constants */
38
39          HashTable const_filenames;
40
41          zend_oparray_context context;
42          zend_file_context file_context;
43
44          zend_arena *arena;
45
46          zend_string *empty_string;
47          zend_string *one_char_string[256];
48          zend_string **known_strings;
49          uint32_t    known_strings_count;
50
51          HashTable interned_strings;
52
53          const zend_encoding **script_encoding_list;
54          size_t script_encoding_list_size;
55          zend_bool multibyte;
56          zend_bool detect_unicode;
57          zend_bool encoding_declared;
58
59          zend_ast *ast;
60          zend_arena *ast_arena;
61
62          zend_stack delayed_oplines_stack;
63
64  #ifdef ZTS
65          zval **static_members_table;
66          int last_static_member;
67  #endif
68  };
```

Ukážka kódu B.3: Štruktúra class entry v PHP 7

```
1   typedef struct _zend_class_entry     zend_class_entry;
2
3   struct _zend_class_entry {
4           char type;
5           zend_string *name;
```

```
6         struct _zend_class_entry *parent;
7         int refcount;
8         uint32_t ce_flags;
9
10        int default_properties_count;
11        int default_static_members_count;
12        zval *default_properties_table;
13        zval *default_static_members_table;
14        zval *static_members_table;
15        HashTable function_table;
16        HashTable properties_info;
17        HashTable constants_table;
18
19        union _zend_function *constructor;
20        union _zend_function *destructor;
21        union _zend_function *clone;
22        union _zend_function *__get;
23        union _zend_function *__set;
24        union _zend_function *__unset;
25        union _zend_function *__isset;
26        union _zend_function *__call;
27        union _zend_function *__callstatic;
28        union _zend_function *__tostring;
29        union _zend_function *__debugInfo;
30        union _zend_function *serialize_func;
31        union _zend_function *unserialize_func;
32
33        zend_class_iterator_funcs iterator_funcs;
34
35        /* handlers */
36        zend_object* (*create_object)(zend_class_entry
            *class_type);
37        zend_object_iterator
            *(*get_iterator)(zend_class_entry *ce, zval
            *object, int by_ref);
38        int (*interface_gets_implemented)(zend_class_entry
            *iface, zend_class_entry *class_type);      /*
            a class implements this interface */
39        union _zend_function
```

```
            *(*get_static_method)(zend_class_entry *ce,
            zend_string* method);
40
41          /* serializer callbacks */
42          int (*serialize)(zval *object, unsigned char
               **buffer, size_t *buf_len, zend_serialize_data
               *data);
43          int (*unserialize)(zval *object, zend_class_entry
               *ce, const unsigned char *buf, size_t buf_len,
               zend_unserialize_data *data);
44
45          uint32_t num_interfaces;
46          uint32_t num_traits;
47          zend_class_entry **interfaces;
48
49          zend_class_entry **traits;
50          zend_trait_alias **trait_aliases;
51          zend_trait_precedence **trait_precedences;
52
53          union {
54                  struct {
55                          zend_string *filename;
56                          uint32_t line_start;
57                          uint32_t line_end;
58                          zend_string *doc_comment;
59                  } user;
60                  struct {
61                          const struct _zend_function_entry
                               *builtin_functions;
62                          struct _zend_module_entry *module;
63                  } internal;
64          } info;
65  };
```

Ukážka kódu B.4: Štruktúra zend function v PHP 7

```
1   typedef union  _zend_function        zend_function;
2
3   union _zend_function {
4           zend_uchar type;                                    /*
```

```
            MUST be the first element of this struct! */
5           uint32_t   quick_arg_flags;
6
7           struct {
8               zend_uchar type;  /* never used */
9               zend_uchar arg_flags[3];                    /*
                    bitset of arg_info.pass_by_reference */
10              uint32_t fn_flags;
11              zend_string *function_name;
12              zend_class_entry *scope;
13              union _zend_function *prototype;
14              uint32_t num_args;
15              uint32_t required_num_args;
16              zend_arg_info *arg_info;
17          } common;
18
19          zend_op_array op_array;
20          zend_internal_function internal_function;
21  };
```

Ukážka kódu B.5: Štruktúra zend execute data v PHP 7

```
1   typedef struct _zend_execute_data    zend_execute_data;
2
3   struct _zend_execute_data {
4           const zend_op       *opline;                    /*
              executed opline */
5           zend_execute_data   *call;                      /*
              current call */
6           zval                *return_value;
7           zend_function       *func;                      /*
              executed function */
8           zval                 This;                      /*
              this + call_info + num_args */
9           zend_execute_data   *prev_execute_data;
10          zend_array          *symbol_table;
11  #if ZEND_EX_USE_RUN_TIME_CACHE
12          void               **run_time_cache;            /*
              cache op_array->run_time_cache */
13  #endif
```

```
14 #if ZEND_EX_USE_LITERALS
15         zval                    *literals;                      /*
              cache op_array->literals */
16 #endif
17 };
```