Luka Bele

Seminar pri predmetu Sistemska programska oprema

MENTOR: doc. dr. Tomaž Dobravec

Ljubljana, januar 2025

1 Uvod

V sodobnem svetu visoko-razpoložljive tehnologije so hitri in za delovanje programske opreme nemoteči popravki in posodobitve ključnega pomena. Za integracijo popravkov v že nameščene programe, ki so potencialno tudi sredi izvajanja skrbi krpalnik programske opreme. Poleg splošnih rešitev v poročilu predstavim tudi krpalinke treh najpogostejših operacijskih sistemov. Za ilustracijo konceptov pa implementiram tudi svoj krpalnik binarnih datotek.

2 Sistemsko neodvisna logika

2. 1 Popravek

Spremembo ali dodatek programski opremi imenujemo popravek. To je spremenjen kos kode namenjen programu, katerega želimo spremeniti. Razlogi za to so lahko različni, najpogosteje pa z njimi odpravljamo hrošče, ki se tekom razvoja in uporabe programske opreme zagotovo pojavijo. Poleg tega s krpanjem odpravljamo varnostna tveganja in dodajamo manjše funkcionalnosti. Ločimo lahko spodaj naštete tipe popravkov: [1] [2] [3]

- **Varnostni popravki**

  Zaradi vse večje življenjske dobe in daljšega vzdrževanja programske opreme je neizogibno, da se bodo tekom tega pojavila varnostna tveganja. Ta naslavljajo varnostni popravki, katerih namen je skrb za integriteto sistema, zaupnost podatkov in dosegljivost storitve, njihova lastnost je, da so uvedeni hitro po odkritju varnostnega tveganja. Čas za pripravo popravka pa je odvisen tudi od tega kdo najprej odkrije tveganje in ali je to sporočeno oskrbniku programske opreme ali izrabljeno, po možnosti deljeno z javnostjo s čimer je ogroženih še več uporabnikov.

- **Paketi storitev**

  Poleg odstranitev varnostnih tvegaj in odprave hroščev dostavljajo tudi nove funkcionalnosti, kar predstavlja priročen način za dostavo izboljšav. Za njih je značilno, da njihovo nameščanje vzame več časa, prav tako kot testiranje saj je poskrbljeno, da ne vnesejo novih hroščev ali tveganj v programsko opremo.

- **Hiter ali nujni popravek**

  S tujko imenovan »hotfix«. Usmerjen v odpravo kritičnih varnostnih tveganj in je dostavljen zelo hitro. Kot posledica z manj obsežnim testiranjem, zato mu pogosto sledi še dodatni popravek. Primer nujnega popravka je odprava ranljivosti sistema, ki omogoča neavtoriziran dostop do podatkovne baze.

- **Posodobitve verzije**

  Programsko opremo posodobijo na zadnjo verzijo, z izboljšavami hitrosti, optimizacijo in odpravo manjših hroščev. Niso podvržene tako rigoroznemu testiranju kot nove izdaje. Pri

semantičnemu verzijoniziranju (oblika 1.2.3), vzvratno kompatibilen popravek predstavlja zadnja številka. [4]

- **Začasni popravki**

PTF – Program Temporary Fix označuje začasne popravke predvsem v poslovnih sistemih kjer naslovi npr. začasni popravek procesiranja transakcij. Tem običajno sledijo paketni popravki, ki to težavo naslovijo v celoti.

- **Popravki operacijskega sistema**

Tako kot ostali progami tudi operacijski sistem potrebuje popravke in izboljšave za stabilnost, odpravo varnostnih tveganj. Dostavljeni so v dogovorjenem časovnem obdobju, najboljši primer tega je Microsoftov Patch Tuesday, ko vsak drugi torek v mesecu dostavijo posodobitev za Windows. Takšni popravki vsebujejo posodobitve jedra, gonilnikov in kritičnih sistemskih enot.

- **Posodobitev varnostne programske opreme**

Pod te spadajo posodobitve za antivirusne programe, požarne zidove in sisteme za odkrivanje vdorov. Vsebujejo definicije in načine za odkrivanje najnovejših groženj za operacijski sistem.

V praksi se popravek nanaša na točno določeno verzijo strojne opreme in krpalnik ima funkcijo za prepoznavanje verzije vgrajeno, na podlagi katere distribuira kompatibilni popravek. Krpanje kode omogoča popravljanje že prevedene ali stroje kode, ko izvorna ni na voljo. To je lahko zelo kompleksno saj za to potrebujemo dobro razumevanje ozadja dogajanja, v kakšen jezik se koda prevede itn. Po dostavljanju popravkov programske opreme ne bomo znova prevajali, prav tako je dostavljanje popravkov bolj ekonomično kot prenašanje celotnega vira še enkrat. Popravki se glede na stanje kode vira delijo na binarne in v izvorni kodi. [5]

Binarni popravki

Distribucija popravkov poteka preko podatkovnih datotek, v katerih je vsebina popravka, katero prebere izvršljiv pomožni program. Bajte strojne kode starega programa, ki ga popravlja kar prepiše z novimi. Tu se pojavita dva scenarija, ko je popravek toliko velik kot koda, ki jo nadomešča, v tem primeru stare bajte samo prepiše z novimi. Če pa je popravek

večji kot del programa, ki ga popravlja mora prostor za dodatne bajte najti drugje. V primeru, da je program razdeljen v module, lahko velikost modula samo poveča s premikanjem kazalcev in vstavi bajte na prazna mesta. Če pa to ni mogoče pogosto kodo doda kar na konec datoteke in nastavi skoke in vejitvene ukaze tako, da se ob zagonu najprej izvede nova koda s popravki, ki se razporedi na mesta, kjer so potrebni popravki. Programerji, ki razvijajo strojno opremo morajo na take primere misliti v naprej s tem, da si v naprej v kodi pustijo prostor za kasnejše popravke. Je pa res, da vedno (ali celo v veliki večini primerov) ne vzdržujemo svoje lastne kode ampak jo »podedujemo« od drugih, zato je pogosto iskanje inovativnih rešitev, kot zmanjšanje števila ukazov z izboljšanjem učinkovitosti kode. [1][5]

Popravki izvorne kode

Kadar operiramo z izvorno kodo programa popravke predstavljajo tekstovne razlike med različnimi verzijami datoteke ali tako imenovan diff. Pogosto jih najdemo na odprtokodnih projektih ali pa kot spremembe na skupnem repozitoriju pri delu v skupinah z orodjem za verzioniraje. Po potrjevanju razlik programer svoj vir z izvorno kodo tudi znova prevede. [5]

3 Distribucija popravkov

Spremembe v programski opremi najprej identificiramo s primerjavo različic izvorne kode ali z orodji za binarno analizo. Pri tveganjih je potrebno določiti v katerem delu kode se nahaja in ali je skupno eni ali več knjižnicam. Po tem sledi priprava popravka, ki zajema popravljanje izvorne kode, zamenjavo okvarjenih delov z novimi in ko izvorna datoteka ne obstaja tudi obratni inženiring kode. Tega se lotimo z opazovanjem sistema, s poznavanjem tipa porcesorja lahko strojno kodo zberemo nazaj v ukaze zbirnika kar nam da razumevanje o delovanju programa, to počne razstavljalec. Pripravljenje popravke opremimo z metapodatki kot so ciljni operacijski sistem, odvisnost od drugih podatkov in navodila za integracijo popravka. [6]

Za aplikacije, ki uporabljajo dinamične knjižnice, je pogosto dovolj, da zamenjamo ali posodobimo specifične knjižnice (DLL v Windows ali .so v Linuxu). Ta metoda omogoča večjo modularnost in zmanjšuje potrebo po obsežnih spremembah v glavni aplikaciji. V nekaterih primerih, kjer popravek mora začeti veljati takoj, uporabimo metodo vbrizgavanja kode v

pomnilnik ali ang. in-memory patching, pri čemer se spremenjeni ukazi naložijo neposredno v delovni pomnilnik delujoče aplikacije, kar omogoča uporabo popravka brez ponovnega zagona.

Po integraciji popravka je ključnega pomena preveriti njegovo delovanje in zagotavljanje, da ni povzročil dodatnih težav. Preverjanje vključuje preverjanje integritete datotek s pomočjo kontrolnih vsot ali digitalnih podpisov, ki zagotavljajo, da je bil popravek pravilno uporabljen.

[7]

4 Sistemsko odvisna logika

4.1 Windows Update

Windows Update za posodabljanje programske opreme uporablja štriri osnovne korake: skeniranje, prenos, namestitev in potrditev.

V ozadju orkestrator za posodobitve skenira za posodobitve na strežniku Microsoft Update. Skeniranje se dogaja na naključnih časovnih intervalih, kar preprečuje preobremenjenost strežnika. Posodobitve se ocenijo glede na ustreznost napravi, pri čemer se upoštevajo Microsoftove smernice in skupinske politike.

Windows Update prenese posodobitve v ozadju in jih začasno shrani v lokalno mapo. S sistemom za optimizacijo dostave zagotovi učinkovito rabo pasovne širine, da ne moti drugih aktivnosti.

Ko je prenesena vsebina pripravljena, se prenesejo še metapodatki in vmesna programska oprema za namestitev, ki bo primerjala podatke z naprave in prenesene metapodatke da ustvari nabor akcij. Ta nabor akcij opiše katere datoteke so potrebne od Windos Updatea, in kako začeti posodobitve, ki jih izvaja inštalacijski agent, program Setup.

Po potrebi naredi še ponvni zagon sistema. [8]

5 Ubuntu Linux

V preteklosti je bilo posodabljanje aplikacij in sistemov v Linuxu precej zahtevno in je od uporabnikov zahtevalo delo z ukazno vrstico. Tovrstni proces je bil za nove uporabnike

pogosto preveč kompleksen, kar je posledično pomenilo, da so sistemi ostajali zastareli. Danes se je Linux močno razvil in postal izjemno prijazen do uporabnika, pri čemer veliko nalog deluje samodejno ali prek grafičnega uporabniškega vmesnika.

Ubuntu je znan po svoji uporabniški prijaznosti, kar se kaže tudi v sistemu posodabljanja. Za posodobitve uporablja dva glavna orodja:

- Update Manager: Grafični uporabniški vmesnik za samodejne in ročne posodobitve.
- Apt-get: Zmogljivo orodje za upravljanje paketov prek ukazne vrstice.

Update Manager deluje skoraj povsem samodejno:

- **Varnostne posodobitve** se preverjajo dnevno.
- **Nevarnostne posodobitve** se preverjajo tedensko.

Če so posodobitve na voljo, se prikaže okno, ki uporabniku omogoča pregled in namestitev posodobitev. Koraki za posodobitev:

1. Odpremo System -> Administration -> Update Manager.
2. Kliknemo gumb Check, da preverimo razpoložljive posodobitve.
3. Označimo posodobitve, ki jih želite namestiti, in kliknite Install Updates.
4. Vnesemo skrbniško geslo (sudo) in kliknite OK.
5. Posodobitve se namestijo, pri čemer lahko delo nadaljujete. Nekatere posodobitve zahtevajo ponovni zagon sistema ali odjavo in ponovno prijavo.

Posodabljanje prek ukazne vrstice omogoča popoln nadzor nad paketi, vendar zahteva ročno izvedbo ukaza:

1. Odpremo terminal.
2. Zaženemo ukaz sudo apt-get update za osvežitev seznama posodobitev.
3. Za izvedbo posodobitev zeženemo sudo apt-get upgrade.
4. Pregledamo seznam posodobitev in pritisnite y za potrditev.
5. Počakamo, da se postopek zaključi. [9]

6 Implementacija krpalnika

Izvorna koda:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <libelf.h>
#include <gelf.h>

unsigned long find_function_address(const char *path_to_bin, const char *function){
    unsigned long res;

    // ELF init
    if (elf_version(EV_CURRENT) == EV_NONE){
        fprintf(stderr, "ELF library initialization failed.\n");
        exit(1);
    }

    // Odpremo bin datoteko
    FILE *bin_file = fopen(path_to_bin,"rb");
    if (!bin_file){
        perror("Failed to open binary");
        exit(1);
    }

    // Init strukture, podamo deskriptor
    Elf *elf = elf_begin(fileno(bin_file), ELF_C_READ, NULL);
    if (!elf) {
        fprintf(stderr, "Failed to read ELF file: %s\n", elf_errmsg(-1));
        fclose(bin_file);
        exit(1);
    }

    // Podamo indeks tabele headerjev
    size_t shstrndx;
    if (elf_getshdrstrndx(elf, &shstrndx) < 0){
```

```
    fprintf(stderr, "Failed to get section header string index: %s\n", elf_errmsg(-1));
    elf_end(elf);
    fclose(bin_file);
    exit(1);
}

Elf_Scn *section = NULL;
GElf_Shdr shdr;

// Iteracija cez sekcije
while ((section = elf_nextscn(elf, section)) != NULL){
    // Beremo zaglavje trenutne sekcije
    if (gelf_getshdr(section, &shdr) != &shdr){
        fprintf(stderr, "Failed to read section header\n");
        elf_end(elf);
        fclose(bin_file);
        exit(1);
    }
    // Ali smo v tabeli simbolov? (Staticna || V izvajanju), preberemo simbole
    if (shdr.sh_type == SHT_SYMTAB || shdr.sh_type == SHT_DYNSYM){
        Elf_Data *data = elf_getdata(section, NULL);
        if (!data){
            fprintf(stderr, "Failed to get section data\n");
            elf_end(elf);
            fclose(bin_file);
            exit(1);
        }
        // Beremo simbole, primerjamo z vhodnim nizom, vrnemo naslov pravega simbola
        int symbols = shdr.sh_size / shdr.sh_entsize;
        for (int i = 0; i < symbols; i++){
            GElf_Sym sym;
            if (gelf_getsym(data, i, &sym) != &sym){
                fprintf(stderr, "Failed to get symbol\n");
                continue;
            }

            const char *name = elf_strptr(elf, shdr.sh_link, sym.st_name);
            if (name && strcmp(name, function) == 0){
```

```
                    printf("Symbol VA: 0x%lx\n", sym.st_value);
                    elf_end(elf);
                    fclose(bin_file);
                    return sym.st_value;
                }
            }
        }
    }

    fprintf(stderr, "Function %s not found in the binary.\n", function);
    elf_end(elf);
    fclose(bin_file);
}

void patch(unsigned char *bufPtr){
    for(size_t i = 0; i < 5; i++){
        bufPtr[i] = 0x90;
    }
}

int main(int argc, char *argv[]){
    if (argc != 3) {
        printf("Run with arguments: %s <binary> <function_name>\n", argv[0]);
        exit(1);
    }


    size_t res = (size_t)find_function_address(argv[1], argv[2]);
    //printf("%zu\n", res);

    const char *filename = argv[1];
    FILE *programfile = fopen(filename, "rb");
    if (!programfile){
        perror("Failed to open");
        return EXIT_FAILURE;
    }

    //Določim velikost vhodne datoteke
```

```
fseek(programfile, 0, SEEK_END);
long size = ftell(programfile);
rewind(programfile);

unsigned char *buffer = (unsigned char *)malloc(size);
if (!buffer) {
    perror("Memory allocation failed");
    fclose(programfile);
    exit(1);
}

size_t bytes_read = fread(buffer, 1, size, programfile);
if (bytes_read != size) {
    perror("Error reading file");
    free(buffer);
    fclose(programfile);
    exit(1);
}

//Izpišemo progam v bajtih
for (size_t i = res; i < (size_t)size; i++) {
    if(buffer[i] == 0xe8){
        patch(&buffer[i]);
        break;
    }
}

// Write the modified buffer to a new file "b.out"
FILE *output_file = fopen("b.out", "wb");
if (!output_file) {
    perror("Failed to open output file");
    free(buffer);
    fclose(programfile);
    exit(1);
}

size_t bytes_written = fwrite(buffer, 1, size, output_file);
if (bytes_written != size) {
```

```
        perror("Error writing to output file");
        free(buffer);
        fclose(programfile);
        fclose(output_file);
        exit(1);
    }

    printf("Patched binary saved to b.out\n");

    free(buffer);
    fclose(programfile);
    return 0;
}
```

7 Viri

[1] J. DADZIE, „ACM - Understanding Software Patching,“ marec 2005. [Elektronski]. Available: https://dl.acm.org/doi/pdf/10.1145/1053331.1053343. [Poskus dostopa 14 januar 2025].

[2] „Glosbe,“ [Elektronski]. Available: https://app.glosbe.com/translation?id=-7758986572258171044. [Poskus dostopa 14 januar 2025].

[3] S. Naveed, „Software Patches: 7 Types of Patches to Remember,“ Pureversity, [Elektronski]. Available: https://www.pureversity.com/blog/types-of-patches. [Poskus dostopa 14 januar 2025].

[4] T. Preston-Werner, „Semantic Versioning,“ [Elektronski]. Available: https://semver.org/. [Poskus dostopa 14 januar 2025].

[5] „Patch (computing),“ [Elektronski]. Available: https://www.wikiwand.com/en/articles/Patch_%28computing%29#Binary_patches. [Poskus dostopa 14 januar 2025].

[6] B. Lutkevich, „Reverse engineering,“ junij 2021. [Elektronski]. Available: https://www.techtarget.com/searchsoftwarequality/definition/reverse-engineering. [Poskus dostopa 14 januar 2025].

[7] S. Rothlisberger, „AMSI Bypass Memory Patch Technique in 2024,“ 24 januar 2024. [Elektronski]. Available: https://medium.com/@sam.rothlisberger/amsi-bypass-memory-patch-technique-in-2024-f5560022752b.

[8] Microsoft, „How Windows Update works,“ 2019. [Elektronski]. Available: https://learn.microsoft.com/en-us/windows/deployment/update/how-windows-update-works#installing-updates.

[9] T. L. Foundation, „Classic SysAdmin: Linux 101: Updating Your System,“ 9 april 2022. [Elektronski]. Available: https://www.linuxfoundation.org/blog/blog/classic-sysadmin-linux-101-updating-your-system.