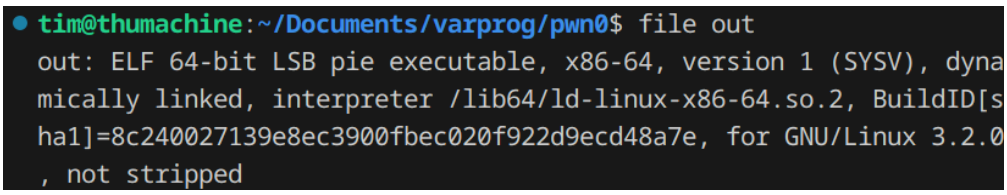
Binary exploitation:

- ko je npr. segmentation fault, mogoče lahko program pripravimo, da z nekim inputom izvaja neke druge zadeve

Web zadeve:

- log4j - logger za Javo je imel vulnerability, da je dvakrat parsal nekaj
- rezultat gcc-ja je `out` datoteka - to je binary
- OS pogleda tip binary fila in ga zalaufa na pravilen način\
- 

- to je nek mesh vseh informacij skupaj, kaj je po linkano
- `objdump -d out` nam ven vrže assembly

```c
#include <stdio.h>

int main() {
    printf("Hello\n");
    return 0;
}
```

.text je prva stvar, ki se zažene

- v disassemblyju ne vidimo printf, ker ne delamo nobenega formattinga - rabimo samo `puts`
- preden gremo v main se zalaufa `__libc_start_main()` - to je runtime za laufat naš program - to nam zalaufa glavne zadeve, ki jih rabimo; to je prva stvar, ki se zalaufa v `_start`
- `_start` je edina stvar, ki jo res nujno rabimo
- na koncu main, rabiš klicat `exit()`
- RDI je prvi parameter za funkcijo
- %n pri `printf` je koliko stvari, sem do sedaj zapisal noter, to zapiše na podani naslov - s `printf` lahko tudi pišemo

```c
#include <stdio.h>

int main() {
```

```
    int a = 10;
    printf("%d let starosti\n%n", a, &a);
    printf("%d\n", a);
    return 0;
}
```

- `scanf`

```
#include <stdio.h>

int main() {
    int a = 10;
    scanf("%d", &a);
    printf("%d let starosti\n%n", a, &a);
    printf("%d\n", a);
    return 0;
}
```

- ko deklariramo spremeljivko, se samo rezervira prostor, nič se ne inicializira
- spomin je iz segmentov:
  - readonly del spomina, readwrite
  - `.text`
  - `_start`
  - `heap`
  - `stack`
  - če pišemo nekam čez rezerviran del, nismo zadeli nobenega segmenta => segmentation fault

```
#include <stdio.h>

int main() {
    char *str;
    scanf("%s", str);
    printf("%s\n", str);
    return 0;
}
```

```
#include <stdio.h>

int main() {
    char *str;
    scanf("%s", &str);
```

```
    printf("%s\n", str);
    return 0;
}
```

- če želimo več prostora za input

```
#include <stdio.h>

int main() {
    char str[10];
    scanf("%s", &str);
    printf("%s\n", str);
    return 0;
}
```

- dobimo stack smashing detected - `-fno-stack-protector` izklopi to in bomo dobili spet segmentation fault, `-no-pie`
- frame buffer - si piše return addrese za vsak klic funkcije
- če v tem primeru preveč stvari napišemo, si bomo povozili return address in bo skočil nekam na random v spominu - lahko točno vemo, kam bo skočil in damo tja neko svojo kodo, ki se bo normalno izvajala naprej
- lahko povozimo return address z npr. naslovom od main in potem bo program po koncu main-a, skočil spet nazaj na začetek main-a

```
#include <stdio.h>

int main() {
    char str[10];
    scanf("%s", &str);
    printf(str);
    return 0;
}
```

- tu se printf kliče z drugačnimi argumenti, prvi argument je format, torej če damo za input `%p%p%p` dobimo random stvari iz stacka na izpisu, `%10$p` pomeni enajsti pointer (štejemo od 0) - lahko beremo arbitrary stvari na stacku in krademo kakšne credentiale, če so tam
- če damo input `%.10f%n`, lahko zapišemo neko random cifro na random memory - lahko injectamo kodo
- `pwndgb` - `gbd` ti pokaže tako, da lažje vidiš kaj se doaja
- `pwndbg out`, `r`, ko damo prevelik input bomo videli stanja programa, ko je crashal
- `ret` vzame naslov iz stacka in gre naprej izvajati na tistem naslovu

- ret = `pop rbp`
- leave = `pop rip`