Lec08: Advanced ROP

Taesoo Kim

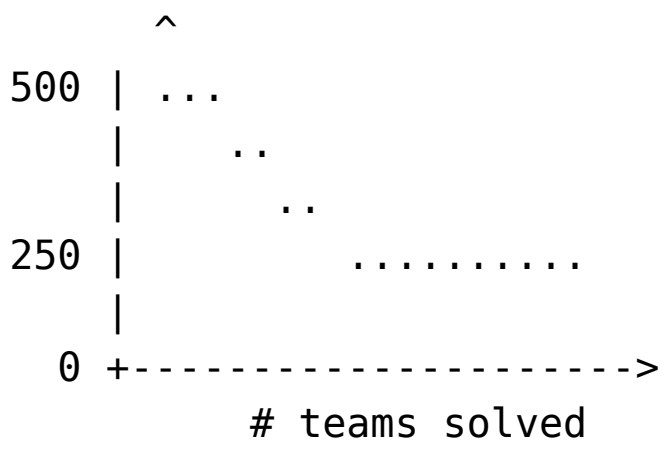
Administrivia

- In-class CTF: https://ctf.gts3.org/ (Open to public! Dec 4)
  - Registration: https://tinyurl.com/tkctf26 (#3-4 persons per team)
  - Rules: https://tc.gts3.org/cs6265/2026-fall/ctf.html
  - Submit your team's challenge by Nov 20
- NSA Codebreaker Challenge → Due: Dec 9

Scoring: Solve + Author

Solve (per challenge): up to 500pt x N challenges

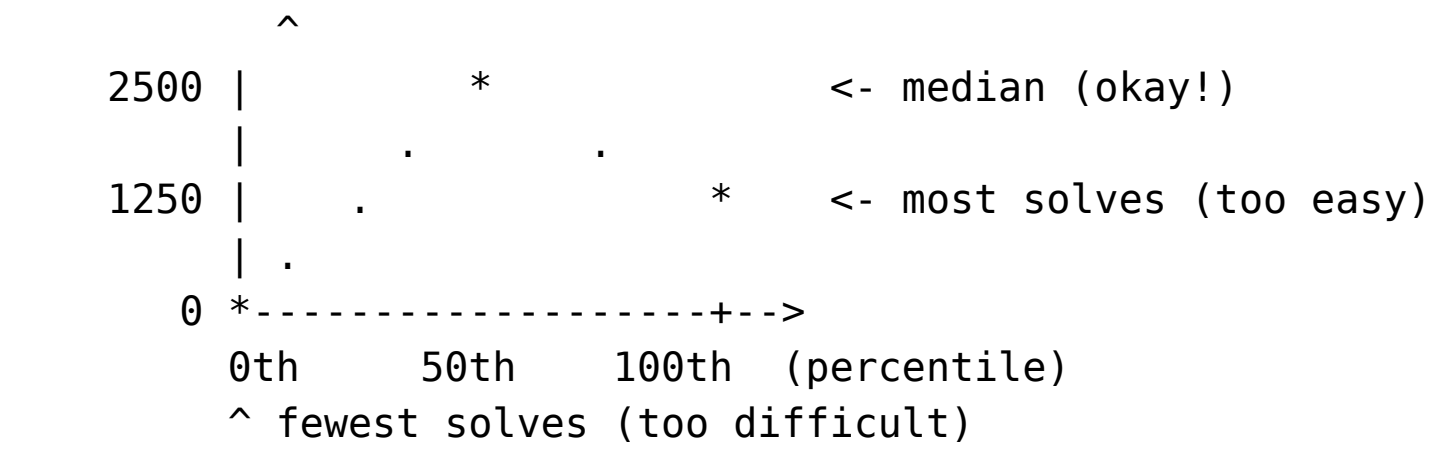
- Dynamic: fewer points as more teams solve it
- Minimum: 250pt (50%)
- Staff challenges: 2x (500-1000pt)



Scoring: Solve + Author

Author: up to 2500pt (per team, about your *own* challenge)

- All challenges (incl. staff) are ranked by # solves
- Linear in between the points below
- Too hard: 0 points, too easy: 1250 points



Overview: CTF Template


```
$ wget https://tc.gts3.org/cs6265/2026-fall/_static/ctf-template.zip
$ unzip ctf-template.zip

$ cd ctf-template
$ make help
prepare: prepare a docker image for compilation
build  : build the target
dist   : build the docker image
release: build the target, docker image and release them
run    : run the docker container
exploit: launch the exploit
test   : test the docker/exploit
submit : zip for submission
```

Today's Tutorial

- In-class tutorial:
  - ROP with no explicit leaks!
  - ROP on x86_64!

Reminder: crackme0x00


```
1  void start() {
2    printf("IOLI Crackme Level 0x00\n");
3    printf("Password:");
4
5    char buf[32];
6    memset(buf, 0, sizeof(buf));
7    read(0, buf, 256);
8
9    if (!strcmp(buf, "250382"))
10     printf("Password OK :)\n");
11   else
12     printf("Invalid Password!\n");
13 }
```

Reminder: crackme0x00


```
$ cat /proc/sys/kernel/randomize_va_space
2

$ checksec ./target
  [*] '/home/lab/tut06-advrop/target'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

→ No shellcode allowed! but not fully randomized.

Reminder: crackme0x00


```c
int main(int argc, char *argv[]) {
  setvbuf(stdout, NULL, _IONBF, 0);
  setvbuf(stdin, NULL, _IONBF, 0);

/* __NO LEAK__!
  void *self = dlopen(NULL, RTLD_NOW);
  printf("stack   : %p\n", &argc);
  printf("system(): %p\n", dlsym(self, "system"));
  printf("printf(): %p\n", dlsym(self, "printf"));
*/

  start();
  return 0;
}
```

→ Then, what would be the first step?

Controlling Arguments in x86_64

Not via stack but via registers!

```
[buf  ]
[.....]
[ra   ] -> pop rdi; ret
[arg1 ]
[ra   ] -> puts()
[ra   ]
```

Leaking libc's Code Pointer

Leaking the libc function stored at `puts@GOT`

```
[buf  ]
[.....]
[ra   ] -> pop rdi; ret
[arg1 ] -> puts@got
[ra   ] -> puts@plt
[ra   ] (crashing)
```

→ We now know libc's memory layout via `puts`! Where to jump next?

Preparing Second Payload

```
[buf  ]
[.....]
[ra   ] -> pop rdi; ret
[arg1 ] -> puts@got
[ra   ] -> puts@plt
[ra   ] -> main
```

→ How does the program behave if we jump to `main` again?

Tutorial Goal: Chaining

```
open("/proc/flag", O_RDONLY)
read(3, tmp, 1040)
write(1, tmp, 1040)
```

→ e.g., open(`rdi` = /proc/flag, `rsi` = O_RDONLY)

In-class Tutorial

- Step1: Controlling arguments in x86_64
- Step2: Leaking libc's code pointer
- Step3: Preparing Second Payload
- Step4: Chaining multiple functions!

```
$ ssh lab06@54.88.195.85
Password: <password>

$ cd tut06-advrop
$ cat README
```

References

- ROP